O'REILLY®

TURING 图灵程序设计丛书

[美] KYLE SIMPSON 著
赵望野 梁杰 译

# 你不知道的JavaScript 上卷

SCOPE & CLOSURES
THIS & OBJECT PROTOTYPES

YOU DON'T KNOW JS

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

# 数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

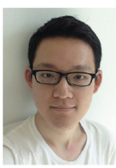
我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

## 译者介绍

### 赵望野

前端工程师，前端基础技术组leader，曾经负责豌豆荚2.0的前端架构设计和主要开发工作，目前负责Front-end Technical Infrastructure的建设。新浪微博@赵望野。负责本书第一部分“作用域和闭包”的翻译。



### 梁杰

北京航空航天大学计算机科学与技术专业大四学生。热爱JavaScript、Python，热爱开源，喜欢做各种各样有趣的事情。负责本书第二部分“this和对象原型”的翻译。

TURING 图灵程序设计丛书

# 你不知道的JavaScript（上卷）

## Scope & Closures
## this & Object Prototypes


［美］Kyle Simpson 著
赵望野　梁杰 译



O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo*

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

你不知道的JavaScript. 上卷 / (美) 辛普森
(Simpson,K.) 著 ; 赵望野, 梁杰译. -- 北京 : 人民邮
电出版社, 2015.4
（图灵程序设计丛书）
ISBN 978-7-115-38573-4

Ⅰ. ①你… Ⅱ. ①辛… ②赵… ③梁… Ⅲ. ①JAVA语
言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第033934号

## 内 容 提 要

很多人对 JavaScript 这门语言的印象都是简单易学，很容易上手。JavaScript 语言本身有很多复杂的概念，语言的使用者不必深入理解这些概念也可以编写出功能全面的应用。殊不知，这些复杂精妙的概念才是语言的精髓，即使是经验丰富的 JavaScript 开发人员，如果没有认真学习的话也无法真正理解它们。在本书中，我们要直面当前 JavaScript 开发者不求甚解的大趋势，深入理解语言内部的机制。

本书既适合 JavaScript 语言初学者阅读，又适合经验丰富的 JavaScript 开发人员深入学习。

◆ 著　　[美] Kyle Simpson
译　　赵望野　梁　杰
责任编辑　李松峰
执行编辑　李　静　曹静雯　魏　然
责任印制　杨林杰
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷
◆ 开本：800×1000　1/16
印张：13
字数：270千字　　　　2015年 4 月第 1 版
印数：1 − 3 500册　　　　2015年 4 月北京第 1次印刷
著作权合同登记号　图字：01-2014-7511号

定价：49.00元
**读者服务热线：(010)51095186转600　印装质量热线：(010)81055316**
**反盗版热线：(010)81055315**
**广告经营许可证：京崇工商广字第 0021 号**

# O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

# 目录

## 第二部分　this 和对象原型

# 前言

在互联网发展的早期，JavaScript 就已经成为了支撑网页内容交互体验的基础技术。那时 JavaScript 的作用可能仅仅是生成一些闪烁的鼠标轨迹或者烦人的弹出窗口，但是经过了大约 20 年的发展，JavaScript 的技术和能力都发生了天翻地覆的变化，现在的 JavaScript 毫无疑问已经成为了世界上使用范围最广的软件平台——互联网——的核心技术。

但是作为一个语言来说，它总是成为大家批评的对象，部分原因是它有很多历史遗留问题，但主要原因是它的设计哲学有问题。就像 Brendan Eich 曾经说过的，JavaScript 甚至连名字都给人一种“蠢弟弟”的感觉，就像是它更成熟的大哥 Java 的不完整版本。不过名字只不过是营销策略上的一个意外，这两个语言有许多本质上的区别。JavaScript 和 Java 的关系，就像 Carnival（嘉年华）和 Car（汽车）的关系一样，八竿子打不着。

JavaScript 借鉴了许多语言的概念和语法，比如 C 风格的过程式编程以及不太明显的 Scheme/List 风格的函数式编程，因此吸引了许多开发者，甚至是那些不会编程的新手。用 JavaScript 来编写“Hello World”是非常简单的，因此这门语言很有吸引力并且很好上手。

虽然 JavaScript 可能是最早出现的语言之一，但是由于其本身的特殊性，相比其他语言，能真正掌握 JavaScript 的人比较少。如果想用 C、C++ 这样的语言编写功能全面的程序，那需要对语言有很深的了解。但是对于 JavaScript 来说，编写功能全面的程序仅仅是冰山一角。

JavaScript 语言本质上有许多复杂的概念，但是却用一种*看起来*比较简单的方式体现出来，比如回调函数，因此 JavaScript 开发者通常只是简单地使用这些特性，并不会关心语言内部的实现原理。

JavaScript 既是一门充满吸引力、简单易用的语言，又是一门具有许多复杂微妙技术的语言，即使是经验丰富的 JavaScript 开发者，如果没有认真学习的话也无法真正理解它们。

这就是 JavaScript 的矛盾之处，也是这门语言的阿喀琉斯之踵[1]。由于 JavaScript 不必理解就可以使用，因此通常来说很难真正理解语言本身，这就是我们面临的挑战。

## 使命

如果每次遇到 JavaScript 中出乎意料的行为时，你的反应就是把它加入黑名单（很多人都是这么做的），那用不了多久你就会把 JavaScript 语言真正的多样性全部排除。

剩下的部分就是非常著名的“好的部分”（Good Parts），但是亲爱的读者们，我恳请你们把它称作“简单的部分”、“安全的部分”甚至“不完整的部分”。

“你不知道的 JavaScript”系列丛书要做的事恰好相反：学习并且深入理解整个 JavaScript，尤其是那些“难的部分”。

在本书中，我们要直面当前 JavaScript 开发者不求甚解的大趋势，他们往往不会深入理解语言内部的机制，遇到困难就会退缩。我们要做的恰好相反，不是退缩，而是继续前进。

你们应当像我一样，不满足于只是让代码正常工作，而是想要弄清楚“为什么”。我希望你能勇于挑战这条崎岖颠簸的“少有人走的路”，拥抱整个 JavaScript。掌握了这些知识之后，无论什么技术、框架和流行词语你都能轻松理解。

这个系列中的每本书专注于语言中一个最容易被误解或者最难理解的核心部分，进行深入、详尽的介绍。在阅读本书时，你应当审视自己对于 JavaScript 的理解，仔细思考书中讲解的理论和那些“你需要知道”的东西。

现在你所理解的 JavaScript 很可能是从别人那里学来的不完整版。这样的 JavaScript 只是真正的 JavaScript 的影子。学完这个系列之后，你就会掌握真正的 JavaScript。读下去吧，我的朋友，JavaScript 恭候你的光临。

## 小结

JavaScript 非常特殊，只学一部分的话非常简单，但是想要完整地学习会很难（就算学到够用也不容易）。当开发者感到迷惑时，他们通常会责怪语言本身，而不是怪自己对语言缺乏了解。这个系列就是为了解决这个问题，让你打心眼儿里欣赏这门语言。

> 本书中的许多例子都需要运行在即将到来的现代 JavaScript 引擎环境中，比如 ES6。部分代码在旧（ES6 之前的）引擎上可能无法正常运行。

注 1：指某人或某事物的最大或者唯一弱点，即罩门关键所在。——译者注

# 本书排版约定

本书中使用以下排版约定。

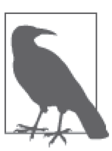
- 楷体
  表示新的术语。

- 等宽字体
  表示代码段以及段落中的程序元素，比如变量、函数名、数据库、数据类型、环境变量、语句以及关键字。

- 等宽粗体
  表示命令中不可改动的部分。

- 等宽斜体
  表示将由用户提供的值（或由上下文确定的值）替换的文本。

这个图标表示提示或建议。



这个图标表示重要说明。

这个图标表示警告或提醒。

# 使用代码示例

可以在这里下载本书第一部分“作用域和闭包”随附的资料（代码示例、练习题等）：http://bit.ly/1c8HEWF。

可以在这里下载本书第二部分“`this` 和对象原型”随附的资料（代码示例、练习题等）：http://bit.ly/ydkjs-this-code

让本书助你一臂之力。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大

段地使用，否则不必与我们联系取得授权。例如，无需请求许可，就可以用本书中的几段代码写成一个程序。但是销售或者发布 O’Reilly 图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处，但不强求。出处信息一般包括书名、作者、出版商和书号，例如：*Scope and Closures*，Kyle Simpson 著（O’Reilly，2014）。版权所有，978-1-491-33558-8。

如果还有关于使用代码的未尽事宜，可以随时与我们联系：permissions@oreilly.com。

## Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O’Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

## 联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书第一部分“作用域和闭包”的网址是 http://oreil.ly/JS_scope_closures。本书第二部分“this 和对象原型”的网址是 http://bit.ly/ydk-js-this-object-prototypes。

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

要查看“你不知道的 JavaScript”系列丛书中的全部图书，请访问：

http://YouDontKnowJS.com

# 第一部分

# 作用域和闭包

［美］Kyle Simpson 著
赵望野 译

# 序

小时候，我特别喜欢把东西拆成零部件，然后再重新装回去——旧的移动电话、立体声音响等我能拿到的一切物件都没能幸免。对于年幼的我来说，使用这些东西还为时过早，但是一旦它们坏了，我就立刻想弄清楚它们的工作原理。

记得有一次，我看见一个老式收音机的电路板，其中有一个缠满铜线的奇怪长管。我不知道这个长管的用途，所以立刻开始研究它。它有什么用？为什么出现在收音机里？为什么它看起来和电路板的其他部分不一样？为什么会有铜线缠绕着它？如果我把铜线拆下来会发生什么？现在我知道了，这是一个在晶体管收音机中很常见的、由缠绕着铜线的铁氧体棒制成的环形天线。

你是否也曾对解答各种各样的为什么很上瘾？大多数孩子都会。事实上，这可能是孩子身上我最喜欢的地方——求知欲很强。

很遗憾，现在我从事着一份专业性的工作，并以制作一些东西来度日。而我儿时的梦想是有一天能够制作那些被我拆开过的东西。当然，现在我所制作的大部分东西都是用 JavaScript 做成的，而不是铁氧体棒……但它们很相似！尽管我曾经一度非常热爱制作东西，但是现在却更渴望了解事物的运行原理。我经常寻找解决问题或修复 bug 的最佳方法，却很少花时间来研究我所使用的工具。

这也是为什么我一看到“你不知道的 JavaScript”系列图书就很激动，因为 JavaScript 的确有很多我不了解的地方。我每天从早到晚都在使用 JavaScript，并且已经持续了好几年，但我真的了解它了吗？答案是否定的。当然，我了解它的很多细节，并且经常阅读标准文档和邮件列表中的内容，但是了解的程度低于我内心那个六岁的孩子希望我达到的水平。

第一部分“作用域和闭包”是一个非常好的切入点。它对于如我一般的受众来说非常有用（希望对你也同样有用）。这本书并不会教你如何使用 JavaScript，但是它会让你意识到对于其内部的运行原理你可能了解得并不多。同时这本书出现的时机也非常巧：ES6 终于稳定下来了，并且各家浏览器的实现工作也正在逐步展开。如果你还没有学习其中的新功能（比如 `let` 和 `const`），这本书将起到很好的介绍作用。

所以我希望你能喜欢这本书，尤其希望 Kyle 对 JavaScript 工作原理每一个细节的批判性思考会渗透到你的思考过程和日常工作中。知其然，也要知其所以然。

Shane Hudson
www.shanehudson.net

# 第 1 章 作用域是什么

几乎所有编程语言最基本的功能之一，就是能够储存变量当中的值，并且能在之后对这个值进行访问或修改。事实上，正是这种储存和访问变量的值的能力将状态带给了程序。

若没有了状态这个概念，程序虽然也能够执行一些简单的任务，但它会受到高度限制，做不到非常有趣。

但是将变量引入程序会引起几个很有意思的问题，也正是我们将要讨论的：这些变量住在哪里？换句话说，它们储存在哪里？最重要的是，程序需要时如何找到它们？

这些问题说明需要一套设计良好的规则来存储变量，并且之后可以方便地找到这些变量。这套规则被称为*作用域*。

但是，究竟在哪里而且怎样设置这些*作用域*的规则呢？

## 1.1 编译原理

尽管通常将 JavaScript 归类为“动态”或“解释执行”语言，但事实上它是一门编译语言。这个事实对你来说可能显而易见，也可能你闻所未闻，取决于你接触过多少编程语言，具有多少经验。但与传统的编译语言不同，它不是提前编译的，编译结果也不能在分布式系统中进行移植。

尽管如此，JavaScript 引擎进行编译的步骤和传统的编译语言非常相似，在某些环节可能比预想的要复杂。

在传统编译语言的流程中，程序中的一段源代码在执行之前会经历三个步骤，统称为“编译”。

- *分词/词法分析*（Tokenizing/Lexing）
  这个过程会将由字符组成的字符串分解成（对编程语言来说）有意义的代码块，这些代码块被称为词法单元（token）。例如，考虑程序 `var a = 2;`。这段程序通常会被分解成为下面这些词法单元：`var`、`a`、`=`、`2` 、`;`。空格是否会被当作词法单元，取决于空格在这门语言中是否具有意义。

> 分词（tokenizing）和词法分析（Lexing）之间的区别是非常微妙、晦涩的，主要差异在于词法单元的识别是通过*有状态*还是*无状态*的方式进行的。简单来说，如果词法单元生成器在判断 `a` 是一个独立的词法单元还是其他词法单元的一部分时，调用的是有状态的解析规则，那么这个过程就被称为*词法分析*。

- *解析/语法分析*（Parsing）
  这个过程是将词法单元流（数组）转换成一个由元素逐级嵌套所组成的代表了程序语法结构的树。这个树被称为“抽象语法树”（Abstract Syntax Tree，AST）。
  `var a = 2;` 的抽象语法树中可能会有一个叫作 `VariableDeclaration` 的顶级节点，接下来是一个叫作 `Identifier`（它的值是 `a`）的子节点，以及一个叫作 `AssignmentExpression` 的子节点。`AssignmentExpression` 节点有一个叫作 `NumericLiteral`（它的值是 `2`）的子节点。

- *代码生成*
  将 AST 转换为可执行代码的过程称被称为代码生成。这个过程与语言、目标平台等息息相关。
  抛开具体细节，简单来说就是有某种方法可以将 `var a = 2;` 的 AST 转化为一组机器指令，用来创建一个叫作 `a` 的变量（包括分配内存等），并将一个值储存在 `a` 中。

> 关于引擎如何管理系统资源超出了我们的讨论范围，因此只需要简单地了解引擎可以根据需要创建并储存变量即可。

比起那些编译过程只有三个步骤的语言的编译器，JavaScript 引擎要复杂得多。例如，在语法分析和代码生成阶段有特定的步骤来对运行性能进行优化，包括对冗余元素进行优化等。

因此在这里只进行宏观、简单的介绍，接下来你就会发现我们介绍的这些看起来有点高深的内容与所要讨论的事情有什么关联。

首先，JavaScript 引擎不会有大量的（像其他语言编译器那么多的）时间用来进行优化，因为与其他语言不同，JavaScript 的编译过程不是发生在构建之前的。

对于 JavaScript 来说，大部分情况下编译发生在代码执行前的几微秒（甚至更短！）的时间内。在我们所要讨论的*作用域*背后，JavaScript 引擎用尽了各种办法（比如 JIT，可以延迟编译甚至实施重编译）来保证性能最佳。

简单地说，任何 JavaScript 代码片段在执行前都要进行编译（通常*就在*执行前）。因此，JavaScript 编译器首先会对 `var a = 2;` 这段程序进行编译，然后做好执行它的准备，并且通常马上就会执行它。

# 1.2 理解作用域

我们学习作用域的方式是将这个过程模拟成几个人物之间的对话。那么，由*谁*进行这场对话呢？

## 1.2.1 演员表

首先介绍将要参与到对程序 `var a = 2;` 进行处理的过程中的演员们，这样才能理解接下来将要听到的对话。

- *引擎*
  从头到尾负责整个 JavaScript 程序的编译及执行过程。

- *编译器*
  引擎的好朋友之一，负责语法分析及代码生成等脏活累活（详见前一节的内容）。

- *作用域*
  引擎的另一位好朋友，负责收集并维护由所有声明的标识符（变量）组成的一系列查询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限。

为了能够*完全理解* JavaScript 的工作原理，你需要开始像引擎（和它的朋友们）一样*思考*，从它们的角度提出问题，并从它们的角度回答这些问题。

## 1.2.2 对话

当你看见 `var a = 2;` 这段程序时，很可能认为这是一句声明。但我们的新朋友引擎却不这么看。事实上，引擎认为这里有两个完全不同的声明，一个由编译器在编译时处理，另一

个则由引擎在运行时处理。

下面我们将 `var a = 2;` 分解，看看引擎和它的朋友们是如何协同工作的。

编译器首先会将这段程序分解成词法单元，然后将词法单元解析成一个树结构。但是当编译器开始进行代码生成时，它对这段程序的处理方式会和预期的有所不同。

可以合理地假设编译器所产生的代码能够用下面的伪代码进行概括："为一个变量分配内存，将其命名为 `a`，然后将值 `2` 保存进这个变量。"然而，这并不完全正确。

事实上编译器会进行如下处理。

1. 遇到 `var a`，编译器会询问作用域是否已经有一个该名称的变量存在于同一个作用域的集合中。如果是，编译器会忽略该声明，继续进行编译；否则它会要求作用域在当前作用域的集合中声明一个新的变量，并命名为 `a`。

2. 接下来编译器会为引擎生成运行时所需的代码，这些代码被用来处理 `a = 2` 这个赋值操作。引擎运行时会首先询问作用域，在当前的作用域集合中是否存在一个叫作 `a` 的变量。如果是，引擎就会使用这个变量；如果否，引擎会继续查找该变量（查看 1.3 节）。

如果引擎最终找到了 `a` 变量，就会将 `2` 赋值给它。否则引擎就会举手示意并抛出一个异常！

**总结**：变量的赋值操作会执行两个动作，首先编译器会在当前作用域中声明一个变量（如果之前没有声明过），然后在运行时引擎会在作用域中查找该变量，如果能够找到就会对它赋值。

## 1.2.3　编译器有话说

为了进一步理解，我们需要多介绍一点编译器的术语。

编译器在编译过程的第二步中生成了代码，引擎执行它时，会通过查找变量 `a` 来判断它是否已声明过。查找的过程由作用域进行协助，但是引擎执行怎样的查找，会影响最终的查找结果。

在我们的例子中，引擎会为变量 `a` 进行 LHS 查询。另外一个查找的类型叫作 RHS。

我打赌你一定能猜到"L"和"R"的含义，它们分别代表左侧和右侧。

什么东西的左侧和右侧？*是一个赋值操作的左侧和右侧。*

换句话说，当变量出现在赋值操作的左侧时进行 LHS 查询，出现在右侧时进行 RHS 查询。

讲得更准确一点，RHS 查询与简单地查找某个变量的值别无二致，而 LHS 查询则是试图找到变量的容器本身，从而可以对其赋值。从这个角度说，RHS 并不是真正意义上的“赋值操作的右侧”，更准确地说是“非左侧”。

你可以将 RHS 理解成 retrieve his source value（取到它的源值），这意味着“得到某某的值”。

让我们继续深入研究。

考虑以下代码：

```
console.log( a );
```

其中对 a 的引用是一个 RHS 引用，因为这里 a 并没有赋予任何值。相应地，需要查找并取得 a 的值，这样才能将值传递给 console.log(..)。

相比之下，例如：

```
a = 2;
```

这里对 a 的引用则是 LHS 引用，因为实际上我们并不关心当前的值是什么，只是想要为 = 2 这个赋值操作找到一个目标。

> LHS 和 RHS 的含义是“赋值操作的左侧或右侧”并不一定意味着就是“= 赋值操作符的左侧或右侧”。赋值操作还有其他几种形式，因此在概念上最好将其理解为“赋值操作的目标是谁（LHS）”以及“谁是赋值操作的源头（RHS）”。

考虑下面的程序，其中既有 LHS 也有 RHS 引用：

```
function foo(a)  {
    console.log( a ); // 2
}

foo( 2 );
```

最后一行 foo(..) 函数的调用需要对 foo 进行 RHS 引用，意味着“去找到 foo 的值，并把它给我”。并且 (..) 意味着 foo 的值需要被执行，因此它最好真的是一个函数类型的值！

这里还有一个容易被忽略却非常重要的细节。

代码中隐式的 a = 2 操作可能很容易被你忽略掉。这个操作发生在 2 被当作参数传递给 foo(..) 函数时，2 会被分配给参数 a。为了给参数 a（隐式地）分配值，需要进行一次 LHS 查询。

这里还有对 a 进行的 RHS 引用，并且将得到的值传给了 console.log(..)。console.log(..) 本身也需要一个引用才能执行，因此会对 console 对象进行 RHS 查询，并且检查得到的值中是否有一个叫作 log 的方法。

最后，在概念上可以理解为在 LHS 和 RHS 之间通过对值 2 进行交互来将其传递进 log(..)（通过变量 a 的 RHS 查询）。假设在 log(..) 函数的原生实现中它可以接受参数，在将 2 赋值给其中第一个（也许叫作 arg1）参数之前，这个参数需要进行 LHS 引用查询。

> 你可能会倾向于将函数声明 function foo(a) {... 概念化为普通的变量声明和赋值，比如 var foo、foo = function(a) {...。如果这样理解的话，这个函数声明将需要进行 LHS 查询。
>
> 然而还有一个重要的细微差别，编译器可以在代码生成的同时处理声明和值的定义，比如在引擎执行代码时，并不会有线程专门用来将一个函数值“分配给”foo。因此，将函数声明理解成前面讨论的 LHS 查询和赋值的形式并不合适。

## 1.2.4　引擎和作用域的对话

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

让我们把上面这段代码的处理过程想象成一段对话，这段对话可能是下面这样的。

> 引擎：我说作用域，我需要为 foo 进行 RHS 引用。你见过它吗？
>
> 作用域：别说，我还真见过，编译器那小子刚刚声明了它。它是一个函数，给你。
>
> 引擎：哥们太够意思了！好吧，我来执行一下 foo。
>
> 引擎：作用域，还有个事儿。我需要为 a 进行 LHS 引用，这个你见过吗？
>
> 作用域：这个也见过，编译器最近把它声名为 foo 的一个形式参数了，拿去吧。
>
> 引擎：大恩不言谢，你总是这么棒。现在我要把 2 赋值给 a。
>
> 引擎：哥们，不好意思又来打扰你。我要为 console 进行 RHS 引用，你见过它吗？
>
> 作用域：咱俩谁跟谁啊，再说我就是干这个。这个我也有，console 是个内置对象。给你。
>
> 引擎：么么哒。我得看看这里面是不是有 log(..)。太好了，找到了，是一个函数。
>
> 引擎：哥们，能帮我再找一下对 a 的 RHS 引用吗？虽然我记得它，但想再确认一次。
>
> 作用域：放心吧，这个变量没有变动过，拿走，不谢。
>
> 引擎：真棒。我来把 a 的值，也就是 2，传递进 log(..)。
>
> ……

### 1.2.5　小测验

检验一下到目前的理解程度。把自己当作引擎，并同作用域进行一次“对话”：

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. 找到其中所有的 LHS 查询。（这里有 3 处！）
2. 找到其中所有的 RHS 查询。（这里有 4 处！）

查看本章小结中的参考答案。

## 1.3　作用域嵌套

我们说过，作用域是根据名称查找变量的一套规则。实际情况中，通常需要同时顾及几个作用域。

当一个块或函数嵌套在另一个块或函数中时，就发生了作用域的嵌套。因此，在当前作用域中无法找到某个变量时，引擎就会在外层嵌套的作用域中继续查找，直到找到该变量，或抵达最外层的作用域（也就是全局作用域）为止。

考虑以下代码：

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

对 `b` 进行的 RHS 引用无法在函数 `foo` 内部完成，但可以在上一级作用域（在这个例子中就是全局作用域）中完成。

因此，回顾一下引擎和作用域之间的对话，会进一步听到：

> 引擎：`foo` 的作用域兄弟，你见过 `b` 吗？我需要对它进行 RHS 引用。
> 作用域：听都没听过，走开。

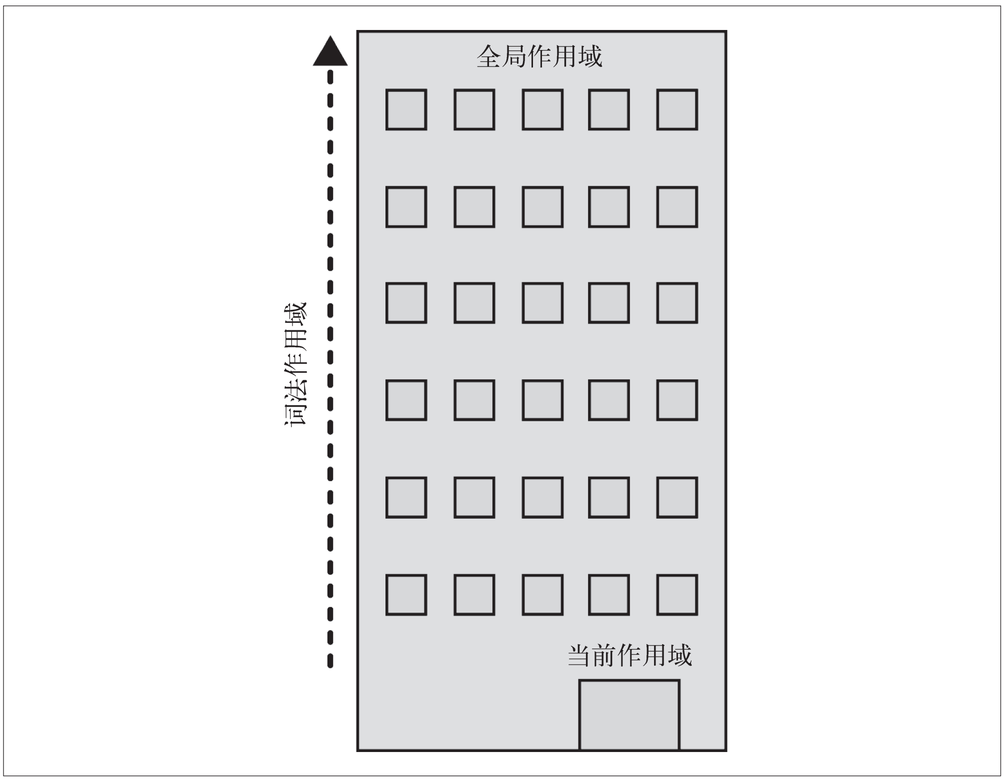
引擎：foo 的上级作用域兄弟，咦？有眼不识泰山，原来你是全局作用域大哥，太好了。你见过 b 吗？我需要对它进行 RHS 引用。

作用域：当然了，给你吧。

遍历嵌套作用域链的规则很简单：引擎从当前的执行作用域开始查找变量，如果找不到，就向上一级继续查找。当抵达最外层的全局作用域时，无论找到还是没找到，查找过程都会停止。

## 把作用域链比喻成一个建筑

为了将作用域处理的过程可视化，我希望你在脑中想象下面这个高大的建筑：




这个建筑代表程序中的嵌套作用域链。第一层楼代表当前的执行作用域，也就是你所处的位置。建筑的顶层代表全局作用域。

LHS 和 RHS 引用都会在当前楼层进行查找，如果没有找到，就会坐电梯前往上一层楼，如果还是没有找到就继续向上，以此类推。一旦抵达顶层（全局作用域），可能找到了你所需的变量，也可能没找到，但无论如何查找过程都将停止。

## 1.4 异常

为什么区分 LHS 和 RHS 是一件重要的事情?

因为在变量还没有声明（在任何作用域中都无法找到该变量）的情况下，这两种查询的行为是不一样的。

考虑如下代码：

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

第一次对 b 进行 RHS 查询时是无法找到该变量的。也就是说，这是一个“未声明”的变量，因为在任何相关的作用域中都无法找到它。

如果 RHS 查询在所有嵌套的作用域中遍寻不到所需的变量，引擎就会抛出 ReferenceError 异常。值得注意的是，ReferenceError 是非常重要的异常类型。

相较之下，当引擎执行 LHS 查询时，如果在顶层（全局作用域）中也无法找到目标变量，全局作用域中就会创建一个具有该名称的变量，并将其返还给引擎，前提是程序运行在非“严格模式”下。

“不，这个变量之前并不存在，但是我很热心地帮你创建了一个。”

ES5 中引入了“严格模式”。同正常模式，或者说宽松 / 懒惰模式相比，严格模式在行为上有很多不同。其中一个不同的行为是严格模式禁止自动或隐式地创建全局变量。因此，在严格模式中 LHS 查询失败时，并不会创建并返回一个全局变量，引擎会抛出同 RHS 查询失败时类似的 ReferenceError 异常。

接下来，如果 RHS 查询找到了一个变量，但是你尝试对这个变量的值进行不合理的操作，比如试图对一个非函数类型的值进行函数调用，或着引用 null 或 undefined 类型的值中的属性，那么引擎会抛出另外一种类型的异常，叫作 TypeError。

ReferenceError 同作用域判别失败相关，而 TypeError 则代表作用域判别成功了，但是对结果的操作是非法或不合理的。

## 1.5 小结

作用域是一套规则，用于确定在何处以及如何查找变量（标识符）。如果查找的目的是对变量进行赋值，那么就会使用 LHS 查询；如果目的是获取变量的值，就会使用 RHS 查询。

赋值操作符会导致 LHS 查询。=操作符或调用函数时传入参数的操作都会导致关联作用域的赋值操作。

JavaScript 引擎首先会在代码执行前对其进行编译，在这个过程中，像 `var a = 2` 这样的声明会被分解成两个独立的步骤：

1. 首先，`var a` 在其作用域中声明新变量。这会在最开始的阶段，也就是代码执行前进行。
2. 接下来，`a = 2` 会查询（LHS 查询）变量 `a` 并对其进行赋值。

LHS 和 RHS 查询都会在当前执行作用域中开始，如果有需要（也就是说它们没有找到所需的标识符），就会向上级作用域继续查找目标标识符，这样每次上升一级作用域（一层楼），最后抵达全局作用域（顶层），无论找到或没找到都将停止。

不成功的 RHS 引用会导致抛出 `ReferenceError` 异常。不成功的 LHS 引用会导致自动隐式地创建一个全局变量（非严格模式下），该变量使用 LHS 引用的目标作为标识符，或者抛出 `ReferenceError` 异常（严格模式下）。

## 小测验答案

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. 找出所有的 LHS 查询（这里有 3 处！）

   *c* = ..;、*a* = 2（隐式变量分配）、*b* = ..

2. 找出所有的 RHS 查询（这里有 4 处！）

   *foo*(2..、= *a*;、*a* ..、.. *b*

# 第 2 章

# 词法作用域

在第 1 章中，我们将“作用域”定义为一套规则，这套规则用来管理引擎如何在当前作用域以及嵌套的子作用域中根据标识符名称进行变量查找。

作用域共有两种主要的工作模型。第一种是最为普遍的，被大多数编程语言所采用的*词法作用域*，我们会对这种作用域进行深入讨论。另外一种叫作*动态作用域*，仍有一些编程语言在使用（比如 Bash 脚本、Perl 中的一些模式等）。

附录 A 中介绍了动态作用域，在这里提到它只是为了同 JavaScript 所采用的作用域模型，即词法作用域模型进行对比。

## 2.1 词法阶段

第 1 章介绍过，大部分标准语言编译器的第一个工作阶段叫作词法化（也叫单词化）。回忆一下，词法化的过程会对源代码中的字符进行检查，如果是有状态的解析过程，还会赋予单词语义。

这个概念是理解词法作用域及其名称来历的基础。

简单地说，词法作用域就是定义在词法阶段的作用域。换句话说，词法作用域是由你在写代码时将变量和块作用域写在哪里来决定的，因此当词法分析器处理代码时会保持作用域不变（大部分情况下是这样的）。

后面会介绍一些欺骗词法作用域的方法，这些方法在词法分析器处理过后依然可以修改作用域，但是这种机制可能有点难以理解。事实上，让词法作用域根据词法关系保持书写时的自然关系不变，是一个非常好的最佳实践。

考虑以下代码：

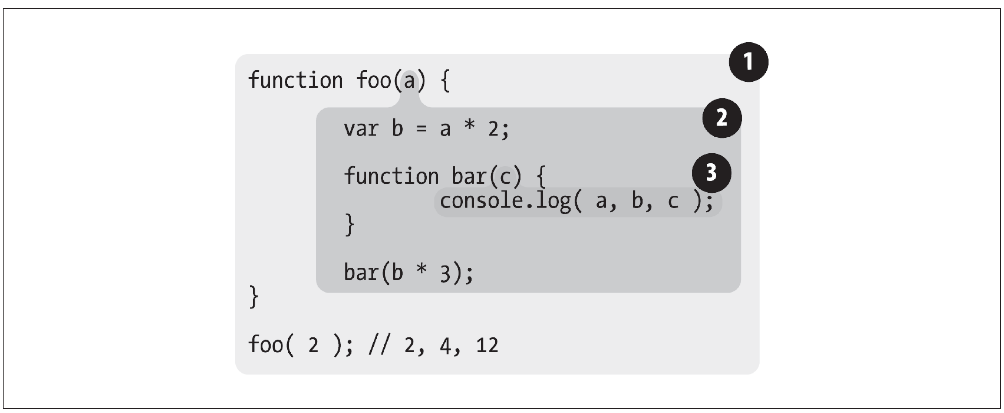
```
function foo(a) {
    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar( b * 3 );
}

foo( 2 ); // 2, 4, 12
```

在这个例子中有三个逐级嵌套的作用域。为了帮助理解，可以将它们想象成几个逐级包含的气泡。




❶ 包含着整个全局作用域，其中只有一个标识符：foo。

❷ 包含着 foo 所创建的作用域，其中有三个标识符：a、bar 和 b。

❸ 包含着 bar 所创建的作用域，其中只有一个标识符：c。

作用域气泡由其对应的作用域块代码写在哪里决定，它们是逐级包含的。下一章会讨论不同类型的作用域，但现在只要假设每一个函数都会创建一个新的作用域气泡就好了。

bar 的气泡被完全包含在 foo 所创建的气泡中，唯一的原因是那里就是我们希望定义函数 bar 的位置。

注意，这里所说的气泡是严格包含的。我们并不是在讨论文氏图[1]这种可以跨越边界的气泡。换句话说，没有任何函数的气泡可以（部分地）同时出现在两个外部作用域的气泡中，就如同没有任何函数可以部分地同时出现在两个父级函数中一样。

## 查找

作用域气泡的结构和互相之间的位置关系给引擎提供了足够的位置信息，引擎用这些信息来查找标识符的位置。

在上一个代码片段中，引擎执行 `console.log(..)` 声明，并查找 `a`、`b` 和 `c` 三个变量的引用。它首先从最内部的作用域，也就是 `bar(..)` 函数的作用域气泡开始查找。引擎无法在这里找到 `a`，因此会去上一级到所嵌套的 `foo(..)` 的作用域中继续查找。在这里找到了 `a`，因此引擎使用了这个引用。对 `b` 来讲也是一样的。而对 `c` 来说，引擎在 `bar(..)` 中就找到了它。

如果 `a`、`c` 都存在于 `bar(..)` 和 `foo(..)` 的内部，`console.log(..)` 就可以直接使用 `bar(..)` 中的变量，而无需到外面的 `foo(..)` 中查找。

*作用域查找会在找到第一个匹配的标识符时停止*。在多层的嵌套作用域中可以定义同名的标识符，这叫作“遮蔽效应”（内部的标识符“遮蔽”了外部的标识符）。抛开遮蔽效应，作用域查找始终从运行时所处的最内部作用域开始，逐级向外或者说向上进行，直到遇见第一个匹配的标识符为止。

> 全局变量会自动成为全局对象（比如浏览器中的 `window` 对象）的属性，因此可以不直接通过全局对象的词法名称，而是间接地通过对全局对象属性的引用来对其进行访问。
>
> ```
> window.a
> ```
>
> 通过这种技术可以访问那些被同名变量所遮蔽的全局变量。但非全局的变量如果被遮蔽了，无论如何都无法被访问到。

无论函数在哪里被调用，也无论它如何被调用，它的词法作用域都*只由函数被声明时所处的位置决定*。

词法作用域查找只会查找一级标识符，比如 `a`、`b` 和 `c`。如果代码中引用了 `foo.bar.baz`，词法作用域查找只会试图查找 `foo` 标识符，找到这个变量后，对象属性访问规则会分别接管对 `bar` 和 `baz` 属性的访问。

---

注 1：集合论中用以表示集合（或类）的一种草图。http://zh.wikipedia.org/wiki/%E6%96%87%E6%B0%8F%E5%9B%BE。——译者注

## 2.2 欺骗词法

如果词法作用域完全由写代码期间函数所声明的位置来定义，怎样才能在运行时来“修改”（也可以说欺骗）词法作用域呢？

JavaScript 中有两种机制来实现这个目的。社区普遍认为在代码中使用这两种机制并不是什么好注意。但是关于它们的争论通常会忽略掉最重要的点：欺骗词法作用域会导致性能下降。

在详细解释性能问题之前，先来看看这两种机制分别是什么原理。

### 2.2.1 eval

JavaScript 中的 eval(..) 函数可以接受一个字符串为参数，并将其中的内容视为好像在书写时就存在于程序中这个位置的代码。换句话说，可以在你写的代码中用程序生成代码并运行，就好像代码是写在那个位置的一样。

根据这个原理来理解 eval(..)，它是如何通过代码欺骗和假装成书写时（也就是词法期）代码就在那，来实现修改词法作用域环境的，这个原理就变得清晰易懂了。

在执行 eval(..) 之后的代码时，引擎并不“知道”或“在意”前面的代码是以动态形式插入进来，并对词法作用域的环境进行修改的。引擎只会如往常地进行词法作用域查找。

考虑以下代码：

```
function foo(str, a) {
    eval( str ); // 欺骗!
    console.log( a, b );
}

var b = 2;

foo( "var b = 3;", 1 ); // 1, 3
```

eval(..) 调用中的 "var b = 3;" 这段代码会被当作本来就在那里一样来处理。由于那段代码声明了一个新的变量 b，因此它对已经存在的 foo(..) 的词法作用域进行了修改。事实上，和前面提到的原理一样，这段代码实际上在 foo(..) 内部创建了一个变量 b，并遮蔽了外部（全局）作用域中的同名变量。

当 console.log(..) 被执行时，会在 foo(..) 的内部同时找到 a 和 b，但是永远也无法找到外部的 b。因此会输出“1, 3”而不是正常情况下会输出的“1, 2”。

在这个例子中，为了展示的方便和简洁，我们传递进去的“代码”字符串是固定不变的。而在实际情况中，可以非常容易地根据程序逻辑动态地将字符拼接在一起之后再传递进去。eval(..) 通常被用来执行动态创建的代码，因为像例子中这样动态地执行一段固定字符所组成的代码，并没有比直接将代码写在那里更有好处。

默认情况下，如果 eval(..) 中所执行的代码包含有一个或多个声明（无论是变量还是函数），就会对 eval(..) 所处的词法作用域进行修改。技术上，通过一些技巧（已经超出我们的讨论范围）可以间接调用 eval(..) 来使其运行在全局作用域中，并对全局作用域进行修改。但无论何种情况，eval(..) 都可以在运行期修改书写期的词法作用域。

在严格模式的程序中，eval(..) 在运行时有其自己的词法作用域，意味着其中的声明无法修改所在的作用域。

```
function foo(str) {
    "use strict";
    eval( str );
    console.log( a ); // ReferenceError: a is not defined
}

foo( "var a = 2" );
```

JavaScript 中还有其他一些功能效果和 eval(..) 很相似。setTimeout(..) 和 setInterval(..) 的第一个参数可以是字符串，字符串的内容可以被解释为一段动态生成的函数代码。这些功能已经过时且并不被提倡。不要使用它们！

new Function(..) 函数的行为也很类似，最后一个参数可以接受代码字符串，并将其转化为动态生成的函数（前面的参数是这个新生成的函数的形参）。这种构建函数的语法比 eval(..) 略微安全一些，但也要尽量避免使用。

在程序中动态生成代码的使用场景非常罕见，因为它所带来的好处无法抵消性能上的损失。

### 2.2.2 with

JavaScript 中另一个难以掌握（并且现在也不推荐使用）的用来欺骗词法作用域的功能是 with 关键字。可以有很多方法来解释 with，在这里我选择从这个角度来解释它：它如何同被它所影响的词法作用域进行交互。

with 通常被当作重复引用同一个对象中的多个属性的快捷方式，可以不需要重复引用对象本身。

比如：

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};

// 单调乏味的重复 "obj"
obj.a = 2;
obj.b = 3;
obj.c = 4;

// 简单的快捷方式
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

但实际上这不仅仅是为了方便地访问对象属性。考虑如下代码：

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};

foo( o1 );
console.log( o1.a ); // 2

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2——不好，a 被泄漏到全局作用域上了！
```

这个例子中创建了 o1 和 o2 两个对象。其中一个具有 a 属性，另外一个没有。foo(..) 函数接受一个 obj 参数，该参数是一个对象引用，并对这个对象引用执行了 with(obj) {..}。在 with 块内部，我们写的代码看起来只是对变量 a 进行简单的词法引用，实际上就是一个 LHS 引用（查看第 1 章），并将 2 赋值给它。

当我们将 o1 传递进去，a = 2 赋值操作找到了 o1.a 并将 2 赋值给它，这在后面的 console.log(o1.a) 中可以体现。而当 o2 传递进去，o2 并没有 a 属性，因此不会创建这个属性，o2.a 保持 undefined。

但是可以注意到一个奇怪的副作用，实际上 `a = 2` 赋值操作创建了一个全局的变量 `a`。这是怎么回事？

`with` 可以将一个没有或有多个属性的对象处理为一个完全隔离的词法作用域，因此这个对象的属性也会被处理为定义在这个作用域中的词法标识符。

> 尽管 `with` 块可以将一个对象处理为词法作用域，但是这个块内部正常的 `var` 声明并不会被限制在这个块的作用域中，而是被添加到 `with` 所处的函数作用域中。

`eval(..)` 函数如果接受了含有一个或多个声明的代码，就会修改其所处的词法作用域，而 `with` 声明实际上是根据你传递给它的对象凭空创建了一个全新的词法作用域。

可以这样理解，当我们传递 `o1` 给 `with` 时，`with` 所声明的作用域是 `o1`，而这个作用域中含有一个同 `o1.a` 属性相符的标识符。但当我们将 `o2` 作为作用域时，其中并没有 `a` 标识符，因此进行了正常的 LHS 标识符查找（查看第 1 章）。

`o2` 的作用域、`foo(..)` 的作用域和全局作用域中都没有找到标识符 `a`，因此当 `a = 2` 执行时，自动创建了一个全局变量（因为是非严格模式）。

`with` 这种将对象及其属性放进一个作用域并同时分配标识符的行为很让人费解。但为了说明我们所看到的现象，这是我能给出的最直白的解释了。

> 另外一个不推荐使用 `eval(..)` 和 `with` 的原因是会被严格模式所影响（限制）。`with` 被完全禁止，而在保留核心功能的前提下，间接或非安全地使用 `eval(..)` 也被禁止了。

## 2.2.3 性能

`eval(..)` 和 `with` 会在运行时修改或创建新的作用域，以此来欺骗其他在书写时定义的词法作用域。

你可能会问，那又怎样呢？如果它们能实现更复杂的功能，并且代码更具有扩展性，难道不是非常好的功能吗？答案是否定的。

JavaScript 引擎会在编译阶段进行数项的性能优化。其中有些优化依赖于能够根据代码的词法进行静态分析，并预先确定所有变量和函数的定义位置，才能在执行过程中快速找到标识符。

但如果引擎在代码中发现了 `eval(..)` 或 `with`，它只能简单地假设关于标识符位置的判断都是无效的，因为无法在词法分析阶段明确知道 `eval(..)` 会接收到什么代码，这些代码会如何对作用域进行修改，也无法知道传递给 `with` 用来创建新词法作用域的对象的内容到底是什么。

最悲观的情况是如果出现了 `eval(..)` 或 `with`，所有的优化可能都是无意义的，因此最简单的做法就是完全不做任何优化。

如果代码中大量使用 `eval(..)` 或 `with`，那么运行起来一定会变得非常慢。无论引擎多聪明，试图将这些悲观情况的副作用限制在最小范围内，也无法避免如果没有这些优化，代码会运行得更慢这个事实。

## 2.3 小结

词法作用域意味着作用域是由书写代码时函数声明的位置来决定的。编译的词法分析阶段基本能够知道全部标识符在哪里以及是如何声明的，从而能够预测在执行过程中如何对它们进行查找。

JavaScript 中有两个机制可以"欺骗"词法作用域：`eval(..)` 和 `with`。前者可以对一段包含一个或多个声明的"代码"字符串进行演算，并借此来修改已经存在的词法作用域（在运行时）。后者本质上是通过将一个对象的引用当作作用域来处理，将对象的属性当作作用域中的标识符来处理，从而创建了一个新的词法作用域（同样是在运行时）。

这两个机制的副作用是引擎无法在编译时对作用域查找进行优化，因为引擎只能谨慎地认为这样的优化是无效的。使用这其中任何一个机制都将导致代码运行变慢。不要使用它们。

第 3 章

# 函数作用域和块作用域

正如我们在第 2 章中讨论的那样，作用域包含了一系列的“气泡”，每一个都可以作为容器，其中包含了标识符（变量、函数）的定义。这些气泡互相嵌套并且整齐地排列成蜂窝型，排列的结构是在写代码时定义的。

但是，究竟是什么生成了一个新的气泡？只有函数会生成新的气泡吗？ JavaScript 中的其他结构能生成作用域气泡吗？

## 3.1 函数中的作用域

对于前面提出的问题，最常见的答案是 JavaScript 具有基于函数的作用域，意味着每声明一个函数都会为其自身创建一个气泡，而其他结构都不会创建作用域气泡。但事实上这并不完全正确，下面我们来看一下。

首先需要研究一下函数作用域及其背后的一些内容。

考虑下面的代码：

```
function foo(a) {
    var b = 2;

    // 一些代码

    function bar() {
        // ...
    }
```

```
        // 更多的代码

        var c = 3;
    }
```

在这个代码片段中，foo(..) 的作用域气泡中包含了标识符 a、b、c 和 bar。无论标识符声明出现在作用域中的何处，这个标识符所代表的变量或函数都将附属于所处作用域的气泡。我们将在下一章讨论具体的原理。

bar(..) 拥有自己的作用域气泡。全局作用域也有自己的作用域气泡，它只包含了一个标识符：foo。

由于标识符 a、b、c 和 bar 都附属于 foo(..) 的作用域气泡，因此无法从 foo(..) 的外部对它们进行访问。也就是说，这些标识符全都无法从全局作用域中进行访问，因此下面的代码会导致 ReferenceError 错误：

```
bar(); // 失败

console.log( a, b, c ); // 三个全都失败
```

但是，这些标识符（a、b、c、foo 和 bar）在 foo(..) 的内部都是可以被访问的，同样在 bar(..) 内部也可以被访问（假设 bar(..) 内部没有同名的标识符声明）。

函数作用域的含义是指，属于这个函数的全部变量都可以在整个函数的范围内使用及复用（事实上在嵌套的作用域中也可以使用）。这种设计方案是非常有用的，能充分利用 JavaScript 变量可以根据需要改变值类型的“动态”特性。

但与此同时，如果不细心处理那些可以在整个作用域范围内被访问的变量，可能会带来意想不到的问题。

## 3.2 隐藏内部实现

对函数的传统认知就是先声明一个函数，然后再向里面添加代码。但反过来想也可以带来一些启示：从所写的代码中挑选出一个任意的片段，然后用函数声明对它进行包装，实际上就是把这些代码“隐藏”起来了。

实际的结果就是在这个代码片段的周围创建了一个作用域气泡，也就是说这段代码中的任何声明（变量或函数）都将绑定在这个新创建的包装函数的作用域中，而不是先前所在的作用域中。换句话说，可以把变量和函数包裹在一个函数的作用域中，然后用这个作用域来“隐藏”它们。

为什么“隐藏”变量和函数是一个有用的技术？

有很多原因促成了这种基于作用域的隐藏方法。它们大都是从最小特权原则中引申出来的，也叫最小授权或最小暴露原则。这个原则是指在软件设计中，应该最小限度地暴露必要内容，而将其他内容都“隐藏”起来，比如某个模块或对象的 API 设计。

这个原则可以延伸到如何选择作用域来包含变量和函数。如果所有变量和函数都在全局作用域中，当然可以在所有的内部嵌套作用域中访问到它们。但这样会破坏前面提到的最小特权原则，因为可能会暴漏过多的变量或函数，而这些变量或函数本应该是私有的，正确的代码应该是可以阻止对这些变量或函数进行访问的。

例如：

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

在这个代码片段中，变量 b 和函数 doSomethingElse(..) 应该是 doSomething(..) 内部具体实现的“私有”内容。给予外部作用域对 b 和 doSomethingElse(..) 的“访问权限”不仅没有必要，而且可能是“危险”的，因为它们可能被有意或无意地以非预期的方式使用，从而导致超出了 doSomething(..) 的适用条件。更“合理”的设计会将这些私有的具体内容隐藏在 doSomething(..) 内部，例如：

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

doSomething( 2 ); // 15
```

现在，b 和 doSomethingElse(..) 都无法从外部被访问，而只能被 doSomething(..) 所控制。功能性和最终效果都没有受影响，但是设计上将具体内容私有化了，设计良好的软件都会依此进行实现。

## 规避冲突

“隐藏”作用域中的变量和函数所带来的另一个好处，是可以避免同名标识符之间的冲突，两个标识符可能具有相同的名字但用途却不一样，无意间可能造成命名冲突。冲突会导致变量的值被意外覆盖。

例如：

```
function foo() {
    function bar(a) {
        i = 3; // 修改 for 循环所属作用域中的 i
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // 糟糕，无限循环了！
    }
}

foo();
```

`bar(..)` 内部的赋值表达式 `i = 3` 意外地覆盖了声明在 `foo(..)` 内部 `for` 循环中的 `i`。在这个例子中将会导致无限循环，因为 `i` 被固定设置为 3，永远满足小于 10 这个条件。

`bar(..)` 内部的赋值操作需要声明一个本地变量来使用，采用任何名字都可以，`var i = 3;` 就可以满足这个需求（同时会为 `i` 声明一个前面提到过的“遮蔽变量”）。另外一种方法是采用一个完全不同的标识符名称，比如 `var j = 3;`。但是软件设计在某种情况下可能自然而然地要求使用同样的标识符名称，因此在这种情况下使用作用域来“隐藏”内部声明是唯一的最佳选择。

### 1. 全局命名空间

变量冲突的一个典型例子存在于全局作用域中。当程序中加载了多个第三方库时，如果它们没有妥善地将内部私有的函数或变量隐藏起来，就会很容易引发冲突。

这些库通常会在全局作用域中声明一个名字足够独特的变量，通常是一个对象。这个对象被用作库的命名空间，所有需要暴露给外界的功能都会成为这个对象（命名空间）的属性，而不是将自己的标识符暴漏在顶级的词法作用域中。

例如：

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
        // ...
```

```
    }
};
```

### 2. 模块管理

另外一种避免冲突的办法和现代的模块机制很接近，就是从众多模块管理器中挑选一个来使用。使用这些工具，任何库都无需将标识符加入到全局作用域中，而是通过依赖管理器的机制将库的标识符显式地导入到另外一个特定的作用域中。

显而易见，这些工具并没有能够违反词法作用域规则的“神奇”功能。它们只是利用作用域的规则强制所有标识符都不能注入到共享作用域中，而是保持在私有、无冲突的作用域中，这样可以有效规避掉所有的意外冲突。

因此，只要你愿意，即使不使用任何依赖管理工具也可以实现相同的功效。第 5 章会介绍模块模式的详细内容。

## 3.3 函数作用域

我们已经知道，在任意代码片段外部添加包装函数，可以将内部的变量和函数定义“隐藏”起来，外部作用域无法访问包装函数内部的任何内容。

例如：

```
var a = 2;

function foo() { // <-- 添加这一行

    var a = 3;
    console.log( a ); // 3

} // <-- 以及这一行
foo(); // <-- 以及这一行

console.log( a ); // 2
```

虽然这种技术可以解决一些问题，但是它并不理想，因为会导致一些额外的问题。首先，必须声明一个具名函数 `foo()`，意味着 `foo` 这个名称本身“污染”了所在作用域（在这个例子中是全局作用域）。其次，必须显式地通过函数名（`foo()`）调用这个函数才能运行其中的代码。

如果函数不需要函数名（或者至少函数名可以不污染所在作用域），并且能够自动运行，这将会更加理想。

幸好，JavaScript 提供了能够同时解决这两个问题的方案、

```
var a = 2;
```

```
(function foo(){ // <-- 添加这一行

    var a = 3;
    console.log( a ); // 3

})(); // <-- 以及这一行

console.log( a ); // 2
```

接下来我们分别介绍这里发生的事情。

首先，包装函数的声明以 (function... 而不仅是以 function... 开始。尽管看上去这并不是一个很显眼的细节，但实际上却是非常重要的区别。函数会被当作函数表达式而不是一个标准的函数声明来处理。

> 区分函数声明和表达式最简单的方法是看 function 关键字出现在声明中的位置（不仅仅是一行代码，而是整个声明中的位置）。如果 function 是声明中的第一个词，那么就是一个函数声明，否则就是一个函数表达式。

函数声明和函数表达式之间最重要的区别是它们的名称标识符将会绑定在何处。

比较一下前面两个代码片段。第一个片段中 foo 被绑定在所在作用域中，可以直接通过 foo() 来调用它。第二个片段中 foo 被绑定在函数表达式自身的函数中而不是所在作用域中。

换句话说，(function foo(){ .. }) 作为函数表达式意味着 foo 只能在 .. 所代表的位置中被访问，外部作用域则不行。foo 变量名被隐藏在自身中意味着不会非必要地污染外部作用域。

### 3.3.1 匿名和具名

对于函数表达式你最熟悉的场景可能就是回调参数了，比如：

```
setTimeout( function() {
    console.log("I waited 1 second!");
}, 1000 );
```

这叫作*匿名函数表达式*，因为 function().. 没有名称标识符。函数表达式可以是匿名的，而函数声明则不可以省略函数名——在 JavaScript 的语法中这是非法的。

匿名函数表达式书写起来简单快捷，很多库和工具也倾向鼓励使用这种风格的代码。但是它也有几个缺点需要考虑。

1. 匿名函数在栈追踪中不会显示出有意义的函数名，使得调试很困难。

2. 如果没有函数名，当函数需要引用自身时只能使用已经过期的 `arguments.callee` 引用，比如在递归中。另一个函数需要引用自身的例子，是在事件触发后事件监听器需要解绑自身。
3. 匿名函数省略了对于代码可读性 / 可理解性很重要的函数名。一个描述性的名称可以让代码不言自明。

行内函数表达式非常强大且有用——匿名和具名之间的区别并不会对这点有任何影响。给函数表达式指定一个函数名可以有效解决以上问题。始终给函数表达式命名是一个最佳实践：

```
setTimeout( function timeoutHandler() { // <-- 快看，我有名字了！

    console.log( "I waited 1 second!" );
}, 1000 );
```

### 3.3.2　立即执行函数表达式

```
var a = 2;

(function foo() {

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

由于函数被包含在一对 ( ) 括号内部，因此成为了一个表达式，通过在末尾加上另外一个 ( ) 可以立即执行这个函数，比如 `(function foo(){ .. })()`。第一个 ( ) 将函数变成表达式，第二个 ( ) 执行了这个函数。

这种模式很常见，几年前社区给它规定了一个术语：IIFE，代表立即执行函数表达式（Immediately Invoked Function Expression）；

函数名对 IIFE 当然不是必须的，IIFE 最常见的用法是使用一个匿名函数表达式。虽然使用具名函数的 IIFE 并不常见，但它具有上述匿名函数表达式的所有优势，因此也是一个值得推广的实践。

```
var a = 2;

(function IIFE() {

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

相较于传统的 IIFE 形式，很多人都更喜欢另一个改进的形式：(function(){ .. }())。仔细观察其中的区别。第一种形式中函数表达式被包含在 ( ) 中，然后在后面用另一个 () 括号来调用。第二种形式中用来调用的 () 括号被移进了用来包装的 ( ) 括号中。

这两种形式在功能上是一致的。选择哪个全凭个人喜好。

IIFE 的另一个非常普遍的进阶用法是把它们当作函数调用并传递参数进去。

例如：

```
var a = 2;

(function IIFE( global ) {

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

console.log( a ); // 2
```

我们将 window 对象的引用传递进去，但将参数命名为 global，因此在代码风格上对全局对象的引用变得比引用一个没有“全局”字样的变量更加清晰。当然可以从外部作用域传递任何你需要的东西，并将变量命名为任何你觉得合适的名字。这对于改进代码风格是非常有帮助的。

这个模式的另外一个应用场景是解决 undefined 标识符的默认值被错误覆盖导致的异常（虽然不常见）。将一个参数命名为 undefined，但是在对应的位置不传入任何值，这样就可以保证在代码块中 undefined 标识符的值真的是 undefined：

```
undefined = true; // 给其他代码挖了一个大坑！绝对不要这样做！

(function IIFE( undefined ) {

    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }

})();
```

IIFE 还有一种变化的用途是倒置代码的运行顺序，将需要运行的函数放在第二位，在 IIFE 执行之后当作参数传递进去。这种模式在 UMD（Universal Module Definition）项目中被广泛使用。尽管这种模式略显冗长，但有些人认为它更易理解。

```
var a = 2;
```

```
(function IIFE( def ) {
    def( window );
})(function def( global ) {

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

});
```

函数表达式 def 定义在片段的第二部分，然后当作参数（这个参数也叫作 def）被传递进 IIFE 函数定义的第一部分中。最后，参数 def（也就是传递进去的函数）被调用，并将 window 传入当作 global 参数的值。

## 3.4　块作用域

尽管函数作用域是最常见的作用域单元，当然也是现行大多数 JavaScript 中最普遍的设计方法，但其他类型的作用域单元也是存在的，并且通过使用其他类型的作用域单元甚至可以实现维护起来更加优秀、简洁的代码。

除 JavaScript 外的很多编程语言都支持块作用域，因此其他语言的开发者对于相关的思维方式会很熟悉，但是对于主要使用 JavaScript 的开发者来说，这个概念会很陌生。

尽管你可能连一行带有块作用域风格的代码都没有写过，但对下面这种很常见的 JavaScript 代码一定很熟悉：

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

我们在 for 循环的头部直接定义了变量 i，通常是因为只想在 for 循环内部的上下文中使用 i，而忽略了 i 会被绑定在外部作用域（函数或全局）中的事实。

这就是块作用域的用处。变量的声明应该距离使用的地方越近越好，并最大限度地本地化。另外一个例子：

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

bar 变量仅在 if 声明的上下文中使用，因此如果能将它声明在 if 块内部中会是一个很有意义的事情。但是，当使用 var 声明变量时，它写在哪里都是一样的，因为它们最终都会

属于外部作用域。这段代码是为了风格更易读而伪装出的形式上的块作用域，如果使用这种形式，要确保没在作用域其他地方意外地使用 bar 只能依靠自觉性。

块作用域是一个用来对之前的最小授权原则进行扩展的工具，将代码从在函数中隐藏信息扩展为在块中隐藏信息。

再次考虑 for 循环的例子：

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

为什么要把一个只在 for 循环内部使用（至少是应该只在内部使用）的变量 i 污染到整个函数作用域中呢？

更重要的是，开发者需要检查自己的代码，以避免在作用范围外意外地使用（或复用）某些变量，如果在错误的地方使用变量将导致未知变量的异常。变量 i 的块作用域（如果存在的话）将使得其只能在 for 循环内部使用，如果在函数中其他地方使用会导致错误。这对保证变量不会被混乱地复用及提升代码的可维护性都有很大帮助。

但可惜，表面上看 JavaScript 并没有块作用域的相关功能。

除非你更加深入地研究。

### 3.4.1 with

我们在第 2 章讨论过 with 关键字。它不仅是一个难于理解的结构，同时也是块作用域的一个例子（块作用域的一种形式），用 with 从对象中创建出的作用域仅在 with 声明中而非外部作用域中有效。

### 3.4.2 try/catch

非常少有人会注意到 JavaScript 的 ES3 规范中规定 try/catch 的 catch 分句会创建一个块作用域，其中声明的变量仅在 catch 内部有效。

例如：

```
try {
    undefined(); // 执行一个非法操作来强制制造一个异常
}
catch (err) {
    console.log( err ); // 能够正常执行!
}

console.log( err ); // ReferenceError: err not found
```

正如你所看到的，err 仅存在 catch 分句内部，当试图从别处引用它时会抛出错误。

> 尽管这个行为已经被标准化，并且被大部分的标准 JavaScript 环境（除了老版本的 IE 浏览器）所支持，但是当同一个作用域中的两个或多个 catch 分句用同样的标识符名称声明错误变量时，很多静态检查工具还是会发出警告。实际上这并不是重复定义，因为所有变量都被安全地限制在块作用域内部，但是静态检查工具还是会很烦人地发出警告。
>
> 为了避免这个不必要的警告，很多开发者会将 catch 的参数命名为 err1、err2 等。也有开发者干脆关闭了静态检查工具对重复变量名的检查。

也许 catch 分句会创建块作用域这件事看起来像教条的学院理论一样没什么用处，但是查看附录 B 就会发现一些很有用的信息。

### 3.4.3 let

到目前为止，我们知道 JavaScript 在暴露块作用域的功能中有一些奇怪的行为。如果仅仅是这样，那么 JavaScript 开发者多年来也就不会将块作用域当作非常有用的机制来使用了。

幸好，ES6 改变了现状，引入了新的 let 关键字，提供了除 var 以外的另一种变量声明方式。

let 关键字可以将变量绑定到所在的任意作用域中（通常是 { .. } 内部）。换句话说，let 为其声明的变量隐式地了所在的块作用域。

```
var foo = true;

if (foo) {
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}

console.log( bar ); // ReferenceError
```

用 let 将变量附加在一个已经存在的块作用域上的行为是隐式的。在开发和修改代码的过程中，如果没有密切关注哪些块作用域中有绑定的变量，并且习惯性地移动这些块或者将其包含在其他的块中，就会导致代码变得混乱。

为块作用域显式地创建块可以部分解决这个问题，使变量的附属关系变得更加清晰。通常来讲，显式的代码优于隐式或一些精巧但不清晰的代码。显式的块作用域风格非常容易书写，并且和其他语言中块作用域的工作原理一致：

```
var foo = true;
```

```
if (foo) {
    { // <-- 显式的快
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
    }
}

console.log( bar ); // ReferenceError
```

只要声明是有效的，在声明中的任意位置都可以使用 `{ .. }` 括号来为 `let` 创建一个用于绑定的块。在这个例子中，我们在 `if` 声明内部显式地创建了一个块，如果需要对其进行重构，整个块都可以被方便地移动而不会对外部 `if` 声明的位置和语义产生任何影响。

> 关于另外一种显式的块作用域表达式的内容，请查看附录 B。

在第 4 章，我们会讨论提升，提升是指声明会被视为存在于其所出现的作用域的整个范围内。

但是使用 `let` 进行的声明不会在块作用域中进行提升。声明的代码被运行之前，声明并不"存在"。

```
{
    console.log( bar ); // ReferenceError!
    let bar = 2;
}
```

### 1. 垃圾收集

另一个块作用域非常有用的原因和闭包及回收内存垃圾的回收机制相关。这里简要说明一下，而内部的实现原理，也就是闭包的机制会在第 5 章详细解释。

考虑以下代码：

```
function process(data) {
    // 在这里做点有趣的事情
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt) {
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

click 函数的点击回调并不需要 someReallyBigData 变量。理论上这意味着当 process(..) 执行后，在内存中占用大量空间的数据结构就可以被垃圾回收了。但是，由于 click 函数形成了一个覆盖整个作用域的闭包，JavaScript 引擎极有可能依然保存着这个结构（取决于具体实现）。

块作用域可以打消这种顾虑，可以让引擎清楚地知道没有必要继续保存 someReallyBigData 了：

```
function process(data) {
    // 在这里做点有趣的事情
}

// 在这个块中定义的内容可以销毁了!
{
    let someReallyBigData = { .. };

    process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

为变量显式声明块作用域，并对变量进行本地绑定是非常有用的工具，可以把它添加到你的代码工具箱中了。

### 2. let循环

一个 let 可以发挥优势的典型例子就是之前讨论的 for 循环。

```
for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError
```

for 循环头部的 let 不仅将 i 绑定到了 for 循环的块中，事实上它将其重新绑定到了循环的每一个迭代中，确保使用上一个循环迭代结束时的值重新进行赋值。

下面通过另一种方式来说明每次迭代时进行重新绑定的行为：

```
{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // 每个迭代重新绑定!
        console.log( i );
    }
}
```

每个迭代进行重新绑定的原因非常有趣，我们会在第 5 章讨论闭包时进行说明。

由于 `let` 声明附属于一个新的作用域而不是当前的函数作用域（也不属于全局作用域），当代码中存在对于函数作用域中 `var` 声明的隐式依赖时，就会有很多隐藏的陷阱，如果用 `let` 来替代 `var` 则需要在代码重构的过程中付出额外的精力。

考虑以下代码：

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    if (baz > bar) {
        console.log( baz );
    }

    // ...
}
```

这段代码可以简单地被重构成下面的同等形式：

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;
    // ...
}

if (baz > bar) {
    console.log( baz );
}
```

但是在使用块级作用域的变量时需要注意以下变化：

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- 移动代码时不要忘了 bar!
        console.log( baz );
    }
}
```

参考附录 B，其中介绍了另外一种块作用域形式，可以用更健壮的方式实现目的，并且写出的代码更易维护和重构。

### 3.4.4　const

除了 `let` 以外，ES6 还引入了 `const`，同样可以用来创建块作用域变量，但其值是固定的（常量）。之后任何试图修改值的操作都会引起错误。

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // 包含在 if 中的块作用域常量

    a = 3; // 正常 !
    b = 4; // 错误 !
}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

## 3.5 小结

函数是 JavaScript 中最常见的作用域单元。本质上，声明在一个函数内部的变量或函数会在所处的作用域中“隐藏”起来，这是有意为之的良好软件的设计原则。

但函数不是唯一的作用域单元。块作用域指的是变量和函数不仅可以属于所处的作用域，也可以属于某个代码块（通常指 `{ .. }` 内部）。

从 ES3 开始，`try/catch` 结构在 `catch` 分句中具有块作用域。

在 ES6 中引入了 `let` 关键字（`var` 关键字的表亲），用来在任意代码块中声明变量。`if (..) { let a = 2; }` 会声明一个劫持了 `if` 的 `{ .. }` 块的变量，并且将变量添加到这个块中。

有些人认为块作用域不应该完全作为函数作用域的替代方案。两种功能应该同时存在，开发者可以并且也应该根据需要选择使用何种作用域，创造可读、可维护的优良代码。

第 4 章

# 提升

到现在为止，你应该已经很熟悉作用域的概念，以及根据声明的位置和方式将变量分配给作用域的相关原理了。函数作用域和块作用域的行为是一样的，可以总结为：任何声明在某个作用域内的变量，都将附属于这个作用域。

但是作用域同其中的变量声明出现的位置有某种微妙的联系，而这个细节正是我们将要讨论的内容。

## 4.1　先有鸡还是先有蛋

直觉上会认为 JavaScript 代码在执行时是由上到下一行一行执行的。但实际上这并不完全正确，有一种特殊情况会导致这个假设是错误的。

考虑以下代码：

```
a = 2;

var a;

console.log( a );
```

你认为 console.log(..) 声明会输出什么呢？

很多开发者会认为是 undefined，因为 var a 声明在 a = 2 之后，他们自然而然地认为变量被重新赋值了，因此会被赋予默认值 undefined。但是，真正的输出结果是 2。

考虑另外一段代码：

```
console.log( a );

var a = 2;
```

鉴于上一个代码片段所表现出来的某种非自上而下的行为特点，你可能会认为这个代码片段也会有同样的行为而输出 2。还有人可能会认为，由于变量 a 在使用前没有先进行声明，因此会抛出 ReferenceError 异常。

不幸的是两种猜测都是不对的。输出来的会是 undefined。

那么到底发生了什么？看起来我们面对的是一个先有鸡还是先有蛋的问题。到底是声明（蛋）在前，还是赋值（鸡）在前？

## 4.2 编译器再度来袭

为了搞明白这个问题，我们需要回顾一下第 1 章中关于编译器的内容。回忆一下，引擎会在解释 JavaScript 代码之前首先对其进行编译。编译阶段中的一部分工作就是找到所有的声明，并用合适的作用域将它们关联起来。第 2 章中展示了这个机制，也正是词法作用域的核心内容。

因此，正确的思考思路是，包括变量和函数在内的所有声明都会在任何代码被执行前首先被处理。

当你看到 var a = 2; 时，可能会认为这是一个声明。但 JavaScript 实际上会将其看成两个声明：var a; 和 a = 2;。第一个定义声明是在编译阶段进行的。第二个赋值声明会被留在原地等待执行阶段。

我们的第一个代码片段会以如下形式进行处理：

```
var a;

a = 2;

console.log( a );
```

其中第一部分是编译，而第二部分是执行。

类似地，我们的第二个代码片段实际是按照以下流程处理的：

```
var a;

console.log( a );

a = 2;
```

因此，打个比方，这个过程就好像变量和函数声明从它们在代码中出现的位置被“移动”到了最上面。这个过程就叫作提升。

换句话说，先有蛋（声明）后有鸡（赋值）。

> 只有声明本身会被提升，而赋值或其他运行逻辑会留在原地。如果提升改变了代码执行的顺序，会造成非常严重的破坏。
>
> ```
> foo();
>
> function foo() {
>     console.log( a ); // undefined
>     var a = 2;
> }
> ```

foo 函数的声明（这个例子还包括实际函数的隐含值）被提升了，因此第一行中的调用可以正常执行。

另外值得注意的是，每个作用域都会进行提升操作。尽管前面大部分的代码片段已经简化了（因为它们只包含全局作用域），而我们正在讨论的 foo(..) 函数自身也会在内部对 var a 进行提升（显然并不是提升到了整个程序的最上方）。因此这段代码实际上会被理解为下面的形式：

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
}

foo();
```

可以看到，函数声明会被提升，但是函数表达式却不会被提升。

```
foo(); // 不是 ReferenceError, 而是 TypeError!

var foo = function bar() {
    // ...
};
```

这段程序中的变量标识符 foo() 被提升并分配给所在作用域（在这里是全局作用域），因此 foo() 不会导致 ReferenceError。但是 foo 此时并没有赋值（如果它是一个函数声明而不是函数表达式，那么就会赋值）。foo() 由于对 undefined 值进行函数调用而导致非法操作，因此抛出 TypeError 异常。

同时也要记住，即使是具名的函数表达式，名称标识符在赋值之前也无法在所在作用域中

使用：

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

这个代码片段经过提升后，实际上会被理解为以下形式：

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

## 4.3　函数优先

函数声明和变量声明都会被提升。但是一个值得注意的细节（这个细节可以出现在有多个“重复”声明的代码中）是函数会首先被提升，然后才是变量。

考虑以下代码：

```
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

foo = function() {
    console.log( 2 );
};
```

会输出 1 而不是 2！这个代码片段会被引擎理解为如下形式：

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
    console.log( 2 );
};
```

注意，var foo 尽管出现在 function foo()... 的声明之前，但它是重复的声明（因此被忽略了），因为函数声明会被提升到普通变量之前。

尽管重复的 var 声明会被忽略掉，但出现在后面的函数声明还是可以覆盖前面的。

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

虽然这些听起来都是些无用的学院理论，但是它说明了在同一个作用域中进行重复定义是非常糟糕的，而且经常会导致各种奇怪的问题。

一个普通块内部的函数声明通常会被提升到所在作用域的顶部，这个过程不会像下面的代码暗示的那样可以被条件判断所控制：

```
foo(); // "b"

var a = true;
if (a) {
    function foo() { console.log("a"); }
}
else {
    function foo() { console.log("b"); }
}
```

但是需要注意这个行为并不可靠，在 JavaScript 未来的版本中有可能发生改变，因此应该尽可能避免在块内部声明函数。

## 4.4　小结

我们习惯将 var a = 2; 看作一个声明，而实际上 JavaScript 引擎并不这么认为。它将 var a 和 a = 2 当作两个单独的声明，第一个是编译阶段的任务，而第二个则是执行阶段的任务。

这意味着无论作用域中的声明出现在什么地方，都将在代码本身被执行前首先进行处理。可以将这个过程形象地想象成所有的声明（变量和函数）都会被“移动”到各自作用域的最顶端，这个过程被称为提升。

声明本身会被提升，而包括函数表达式的赋值在内的赋值操作并不会提升。

要注意避免重复声明，特别是当普通的 var 声明和函数声明混合在一起的时候，否则会引起很多危险的问题！

# 第 5 章
# 作用域闭包

接下来的内容需要对作用域工作原理相关的基础知识有非常深入的理解。

我们将注意力转移到这门语言中一个非常重要但又难以掌握，*近乎神话*的概念上：*闭包*。如果你了解了之前关于词法作用域的讨论，那么闭包的概念几乎是不言自明的。*魔术师的幕布后藏着一个人，我们将要揭开他的伪装*。我可没说这个人是 Crockford[1]！

在继续学习之前，如果你还是对词法作用域相关内容有疑问，可以重新回顾一下第 2 章中的相关内容，现在是个好机会。

## 5.1 启示

对于那些有一点 JavaScript 使用经验但从未真正理解闭包概念的人来说，理解闭包可以看作是某种意义上的重生，但是需要付出非常多的努力和牺牲才能理解这个概念。

回忆我前几年的时光，大量使用 JavaScript 但却完全不理解闭包是什么。总是感觉这门语言有其隐蔽的一面，如果能够掌握将会功力大涨，但讽刺的是我始终无法掌握其中的门道。还记得我曾经大量阅读早期框架的源码，试图能够理解闭包的工作原理。现在还能回忆起我的脑海中第一次浮现出关于“模块模式”相关概念时的激动心情。

那时我无法理解并且倾尽数年心血来探索的，也就是我马上要传授给你的秘诀：JavaScript

注 1：Douglas Crockford 是 Web 开发领域最知名的技术权威之一，ECMA JavaScript 2.0 标准化委员会委员，被 JavaScript 之父 Brendan Eich 称为 JavaScript 界的宗师。——译者注

中闭包无处不在，你只需要能够识别并拥抱它。闭包并不是一个需要学习新的语法或模式才能使用的工具，它也不是一件必须接受像 Luke[2] 一样的原力训练才能使用和掌握的武器。

闭包是基于词法作用域书写代码时所产生的自然结果，你甚至不需要为了利用它们而有意识地创建闭包。闭包的创建和使用在你的代码中随处可见。你缺少的是根据你自己的意愿来识别、拥抱和影响闭包的思维环境。

最后你恍然大悟：原来在我的代码中已经到处都是闭包了，现在我终于能理解它们了。理解闭包就好像 Neo[3] 第一次见到矩阵 [4] 一样。

## 5.2 实质问题

好了，夸张和浮夸的电影比喻已经够多了。

下面是直接了当的定义，你需要掌握它才能理解和识别闭包：

当函数可以记住并访问所在的词法作用域时，就产生了闭包，即使函数是在当前词法作用域之外执行。

下面用一些代码来解释这个定义。

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a ); // 2
    }

    bar();
}

foo();
```

这段代码看起来和嵌套作用域中的示例代码很相似。基于词法作用域的查找规则，函数 `bar()` 可以访问外部作用域中的变量 `a`（这个例子中的是一个 RHS 引用查询）。

这是闭包吗？

技术上来讲，也许是。但根据前面的定义，确切地说并不是。我认为最准确地用来解释 `bar()` 对 `a` 的引用的方法是词法作用域的查找规则，而这些规则只是闭包的一部分。（但却是非常重要的一部分！）

---

注 2：《星球大战》系列电影中的人物。——译者注

注 3：电影《骇客帝国》的主角。——译者注

注 4：电影《骇客帝国》中拥有自我意识主宰一切的超级计算机。——译者注

从纯学术的角度说，在上面的代码片段中，函数 bar() 具有一个涵盖 foo() 作用域的闭包（事实上，涵盖了它能访问的所有作用域，比如全局作用域）。也可以认为 bar() 被封闭在了 foo() 的作用域中。为什么呢？原因简单明了，因为 bar() 嵌套在 foo() 内部。

但是通过这种方式定义的闭包并不能直接进行观察，也无法明白在这个代码片段中闭包是如何工作的。我们可以很容易地理解词法作用域，而闭包则隐藏在代码之后的神秘阴影里，并不那么容易理解。

下面我们来看一段代码，清晰地展示了闭包：

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a );
    }

    return bar;
}

var baz = foo();

baz(); // 2 —— 朋友，这就是闭包的效果。
```

函数 bar() 的词法作用域能够访问 foo() 的内部作用域。然后我们将 bar() 函数本身当作一个值类型进行传递。在这个例子中，我们将 bar 所引用的函数对象本身当作返回值。

在 foo() 执行后，其返回值（也就是内部的 bar() 函数）赋值给变量 baz 并调用 baz()，实际上只是通过不同的标识符引用调用了内部的函数 bar()。

bar() 显然可以被正常执行。但是在这个例子中，它在自己定义的词法作用域以外的地方执行。

在 foo() 执行后，通常会期待 foo() 的整个内部作用域都被销毁，因为我们知道引擎有垃圾回收器用来释放不再使用的内存空间。由于看上去 foo() 的内容不会再被使用，所以很自然地会考虑对其进行回收。

而闭包的“神奇”之处正是可以阻止这件事情的发生。事实上内部作用域依然存在，因此没有被回收。谁在使用这个内部作用域？原来是 bar() 本身在使用。

拜 bar() 所声明的位置所赐，它拥有涵盖 foo() 内部作用域的闭包，使得该作用域能够一直存活，以供 bar() 在之后任何时间进行引用。

bar() 依然持有对该作用域的引用，而这个引用就叫作闭包。

因此，在几微秒之后变量 baz 被实际调用（调用内部函数 bar），不出意料它可以访问定义

时的词法作用域，因此它也可以如预期般访问变量 a。

这个函数在定义时的词法作用域以外的地方被调用。闭包使得函数可以继续访问定义时的词法作用域。

当然，无论使用何种方式对函数类型的值进行传递，当函数在别处被调用时都可以观察到闭包。

```
function foo() {
    var a = 2;

    function baz() {
        console.log( a ); // 2
    }

    bar( baz );
}

function bar(fn) {
    fn(); // 妈妈快看呀，这就是闭包!
}
```

把内部函数 baz 传递给 bar，当调用这个内部函数时（现在叫作 fn），它涵盖的 foo() 内部作用域的闭包就可以观察到了，因为它能够访问 a。

传递函数当然也可以是间接的。

```
var fn;

function foo() {
    var a = 2;

    function baz() {
        console.log( a );
    }

    fn = baz; // 将 baz 分配给全局变量
}

function bar() {
    fn(); // 妈妈快看呀，这就是闭包!
}

foo();

bar(); // 2
```

无论通过何种手段将内部函数传递到所在的词法作用域以外，它都会持有对原始定义作用域的引用，无论在何处执行这个函数都会使用闭包。

## 5.3 现在我懂了

前面的代码片段有点死板，并且为了解释如何使用闭包而人为地在结构上进行了修饰。但我保证闭包绝不仅仅是一个好玩的玩具。你已经写过的代码中一定到处都是闭包的身影。现在让我们来搞懂这个事实。

```
function wait(message) {

    setTimeout( function timer() {
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

将一个内部函数（名为 timer）传递给 setTimeout(..)。timer 具有涵盖 wait(..) 作用域的闭包，因此还保有对变量 message 的引用。

wait(..) 执行 1000 毫秒后，它的内部作用域并不会消失，timer 函数依然保有 wait(..) 作用域的闭包。

深入到引擎的内部原理中，内置的工具函数 setTimeout(..) 持有对一个参数的引用，这个参数也许叫作 fn 或者 func，或者其他类似的名字。引擎会调用这个函数，在例子中就是内部的 timer 函数，而词法作用域在这个过程中保持完整。

这就是闭包。

或者，如果你很熟悉 jQuery（或者其他能说明这个问题的 JavaScript 框架），可以思考下面的代码：

```
function setupBot(name, selector) {
    $( selector ).click( function activator() {
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

我不知道你会写什么样的代码，但是我写的代码负责控制由闭包机器人组成的整个全球无人机大军，这是完全可以实现的！

玩笑开完了，本质上无论何时何地，如果将函数（访问它们各自的词法作用域）当作第一级的值类型并到处传递，你就会看到闭包在这些函数中的应用。在定时器、事件监听器、Ajax 请求、跨窗口通信、Web Workers 或者任何其他的异步（或者同步）任务中，只要使用了回调函数，实际上就是在使用闭包！

> 第 3 章介绍了 IIFE 模式。通常认为 IIFE 是典型的闭包例子，但根据先前对闭包的定义，我并不是很同意这个观点。

```
var a = 2;

(function IIFE() {
    console.log( a );
})();
```

虽然这段代码可以正常工作，但严格来讲它并不是闭包。为什么？因为函数（示例代码中的 IIFE）并不是在它本身的词法作用域以外执行的。它在定义时所在的作用域中执行（而外部作用域，也就是全局作用域也持有 a）。a 是通过普通的词法作用域查找而非闭包被发现的。

尽管技术上来讲，闭包是发生在定义时的，但并不非常明显，就好像六祖慧能所说："既非风动，亦非幡动，仁者心动耳。"[5]。

尽管 IIFE *本身*并不是观察闭包的恰当例子，但它的确创建了闭包，并且也是最常用来创建可以被封闭起来的闭包的工具。因此 IIFE 的确同闭包息息相关，即使本身并不会真的使用闭包。

亲爱的读者，现在把书放下，我有一个任务要给你。打开你最近写的 JavaScript 代码，找到其中的函数类型的值并指出哪里已经使用了闭包，即使你以前可能并不知道这就是闭包。

等你哟！

现在你懂了吧！

## 5.4　循环和闭包

要说明闭包，for 循环是最常见的例子。

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer() {
        console.log( i );
    }, i*1000 );
}
```

注 5：原文为 it's a tree falling in the forest with no one around to hear it，同六祖慧能的风幡之动禅喻近义，比喻客观存在和观察认知之间的关系。——译者注

由于很多开发者对闭包的概念认识得并不是很清楚，因此当循环内部包含函数定义时，代码格式检查器经常发出警告。我们在这里介绍如何才能正确地使用闭包并发挥它的威力，但是代码格式检查器并没有那么灵敏，它会假设你并不真正了解自己在做什么，所以无论如何都会发出警告。

正常情况下，我们对这段代码行为的预期是分别输出数字 1~5，每秒一次，每次一个。

但实际上，这段代码在运行时会以每秒一次的频率输出五次 6。

这是为什么？

首先解释 `6` 是从哪里来的。这个循环的终止条件是 `i` 不再 `<=5`。条件首次成立时 `i` 的值是 6。因此，输出显示的是循环结束时 `i` 的最终值。

仔细想一下，这好像又是显而易见的，延迟函数的回调会在循环结束时才执行。事实上，当定时器运行时即使每个迭代中执行的是 `setTimeout(.., 0)`，所有的回调函数依然是在循环结束后才会被执行，因此会每次输出一个 `6` 出来。

这里引伸出一个更深入的问题，代码中到底有什么缺陷导致它的行为同语义所暗示的不一致呢？

缺陷是我们试图假设循环中的每个迭代在运行时都会给自己“捕获”一个 `i` 的副本。但是根据作用域的工作原理，实际情况是尽管循环中的五个函数是在各个迭代中分别定义的，但是它们都被封闭在一个共享的全局作用域中，因此实际上只有一个 `i`。

这样说的话，当然所有函数共享一个 `i` 的引用。循环结构让我们误以为背后还有更复杂的机制在起作用，但实际上没有。如果将延迟函数的回调重复定义五次，完全不使用循环，那它同这段代码是完全等价的。

下面回到正题。缺陷是什么？我们需要更多的闭包作用域，特别是在循环的过程中每个迭代都需要一个闭包作用域。

第 3 章介绍过，IIFE 会通过声明并立即执行一个函数来创建作用域。

我们来试一下：

```
for (var i=1; i<=5; i++) {
    (function() {
        setTimeout( function timer() {
            console.log( i );
        }, i*1000 );
    })();
}
```

这样能行吗？试试吧，我等着你。

我不卖关子了。这样不行。但是为什么呢？我们现在显然拥有更多的词法作用域了。的确每个延迟函数都会将 IIFE 在每次迭代中创建的作用域封闭起来。

如果作用域是空的，那么仅仅将它们进行封闭是不够的。仔细看一下，我们的 IIFE 只是一个什么都没有的空作用域。它需要包含一点实质内容才能为我们所用。

它需要有自己的变量，用来在每个迭代中储存 i 的值：

```
for (var i=1; i<=5; i++) {
    (function() {
        var j = i;
        setTimeout( function timer() {
            console.log( j );
        }, j*1000 );
    })();
}
```

行了！它能正常工作了！。

可以对这段代码进行一些改进：

```
for (var i=1; i<=5; i++) {
    (function(j) {
        setTimeout( function timer() {
            console.log( j );
        }, j*1000 );
    })( i );
}
```

当然，这些 IIFE 也不过就是函数，因此我们可以将 i 传递进去，如果愿意的话可以将变量名定为 j，当然也可以还叫作 i。无论如何这段代码现在可以工作了。

在迭代内使用 IIFE 会为每个迭代都生成一个新的作用域，使得延迟函数的回调可以将新的作用域封闭在每个迭代内部，每个迭代中都会含有一个具有正确值的变量供我们访问。

问题解决啦！

## 重返块作用域

仔细思考我们对前面的解决方案的分析。我们使用 IIFE 在每次迭代时都创建一个新的作用域。换句话说，每次迭代我们都需要一个块作用域。第 3 章介绍了 let 声明，可以用来劫持块作用域，并且在这个块作用域中声明一个变量。

本质上这是将一个块转换成一个可以被关闭的作用域。因此，下面这些看起来很酷的代码就可以正常运行了：

```
for (var i=1; i<=5; i++) {
    let j = i; // 是的，闭包的块作用域！
    setTimeout( function timer() {
        console.log( j );
    }, j*1000 );
}
```

但是，这还不是全部！（我用 Bob Barker[6] 的声音说道）for 循环头部的 let 声明还会有一个特殊的行为。这个行为指出变量在循环过程中不止被声明一次，每次迭代都会声明。随后的每个迭代都会使用上一个迭代结束时的值来初始化这个变量。

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer() {
        console.log( i );
    }, i*1000 );
}
```

很酷是吧？块作用域和闭包联手便可天下无敌。不知道你是什么情况，反正这个功能让我成为了一名快乐的 JavaScript 程序员。

## 5.5 模块

还有其他的代码模式利用闭包的强大威力，但从表面上看，它们似乎与回调无关。下面一起来研究其中最强大的一个：模块。

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

正如在这段代码中所看到的，这里并没有明显的闭包，只有两个私有数据变量 something 和 another，以及 doSomething() 和 doAnother() 两个内部函数，它们的词法作用域（而这就是闭包）也就是 foo() 的内部作用域。

接下来考虑以下代码：

```
function CoolModule() {
    var something = "cool";
```

注 6：Bob Barker 是美国著名的电视节目主持人。——译者注

```
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

这个模式在 JavaScript 中被称为*模块*。最常见的实现模块模式的方法通常被称为*模块暴露*，这里展示的是其变体。

我们仔细研究一下这些代码。

首先，`CoolModule()` 只是一个函数，必须要通过*调用*它来创建一个模块实例。如果不执行外部函数，内部作用域和闭包都无法被创建。

其次，`CoolModule()` 返回一个用对象字面量语法 `{ key: value, ... }` 来表示的对象。这个返回的对象中含有对内部函数*而不是*内部数据变量的引用。我们保持内部数据变量是隐藏且私有的状态。可以将这个对象类型的返回值看作本质上是*模块的公共* API。

这个对象类型的返回值最终被赋值给外部的变量 `foo`，然后就可以通过它来访问 API 中的属性方法，比如 `foo.doSomething()`。

> 从模块中返回一个实际的对象并不是必须的，也可以直接返回一个内部函数。jQuery 就是一个很好的例子。`jQuery` 和 `$` 标识符就是 jQuery 模块的公共 API，但它们本身都是函数（由于函数也是对象，它们本身也可以拥有属性）。

`doSomething()` 和 `doAnother()` 函数具有涵盖模块实例内部作用域的闭包（通过调用 `CoolModule()` 实现）。当通过返回一个含有属性引用的对象的方式来将函数传递到词法作用域外部时，我们已经创造了可以观察和实践闭包的条件。

如果要更简单的描述，模块模式需要具备两个必要条件。

1. 必须有外部的封闭函数，该函数必须至少被调用一次（每次调用都会创建一个新的模块实例）。
2. 封闭函数必须返回至少一个内部函数，这样内部函数才能在私有作用域中形成闭包，并且可以访问或者修改私有的状态。

一个具有函数属性的对象本身并不是*真正*的模块。从方便观察的角度看，一个从函数调用所返回的，只有数据属性而没有闭包函数的对象并不是*真正*的模块。

上一个示例代码中有一个叫作 `CoolModule()` 的独立的模块创建器，可以被调用任意多次，每次调用都会创建一个新的模块实例。当只需要一个实例时，可以对这个模式进行简单的改进来实现单例模式：

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

我们将模块函数转换成了 IIFE（参见第 3 章），*立即*调用这个函数并将返回值直接赋值给单例的模块实例标识符 `foo`。

模块也是普通的函数，因此可以接受参数：

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );
```

```
foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

模块模式另一个简单但强大的变化用法是，命名将要作为公共 API 返回的对象：

```
var foo = (function CoolModule(id) {
    function change() {
        // 修改公共 API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }

    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

通过在模块实例的内部保留对公共 API 对象的内部引用，可以从内部对模块实例进行修改，包括添加或删除方法和属性，以及修改它们的值。

### 5.5.1　现代的模块机制

大多数模块依赖加载器 / 管理器本质上都是将这种模块定义封装进一个友好的 API。这里并不会研究某个具体的库，为了宏观了解我会简单地介绍一些核心概念：

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }
```

```
    return {
        define: define,
        get: get
    };
})();
```

这段代码的核心是 modules[name] = impl.apply(impl, deps)。为了模块的定义引入了包装函数（可以传入任何依赖），并且将返回值，也就是模块的 API，储存在一个根据名字来管理的模块列表中。

下面展示了如何使用它来定义模块：

```
MyModules.define( "bar", [], function() {
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar) {
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```



"foo" 和 "bar" 模块都是通过一个返回公共 API 的函数来定义的。"foo" 甚至接受 "bar" 的示例作为依赖参数，并能相应地使用它。

为我们自己着想，应该多花一点时间来研究这些示例代码并完全理解闭包的作用吧。最重要的是要理解模块管理器没有任何特殊的“魔力”。它们符合前面列出的模块模式的两个特点：为函数定义引入包装函数，并保证它的返回值和模块的 API 保持一致。

换句话说，模块就是模块，即使在它们外层加上一个友好的包装工具也不会发生任何变化。

## 5.5.2 未来的模块机制

ES6 中为模块增加了一级语法支持。但通过模块系统进行加载时，ES6 会将文件当作独立的模块来处理。每个模块都可以导入其他模块或特定的 API 成员，同样也可以导出自己的 API 成员。

> 基于函数的模块并不是一个能被稳定识别的模式（编译器无法识别），它们的 API 语义只有在运行时才会被考虑进来。因此可以在运行时修改一个模块的 API（参考前面关于公共 API 的讨论）。
>
> 相比之下，ES6 模块 API 更加稳定（API 不会在运行时改变）。由于编辑器知道这一点，因此可以在（的确也这样做了）编译期检查对导入模块的 API 成员的引用是否真实存在。如果 API 引用并不存在，编译器会在运行时抛出一个或多个“早期”错误，而不会像往常一样在运行期采用动态的解决方案。

ES6 的模块没有“行内”格式，必须被定义在独立的文件中（一个文件一个模块）。浏览器或引擎有一个默认的“模块加载器”（可以被重载，但这远超出了我们的讨论范围）可以在导入模块时异步地加载模块文件。

考虑以下代码：

bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

foo.js

```
// 仅从 "bar" 模块导入 hello()
import hello from "bar";

var hungry = "hippo";

function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;
```

baz.js

```
// 导入完整的 "foo" 和 "bar" 模块
```

```
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

> 需要用前面两个代码片段中的内容分别创建文件 foo.js 和 bar.js。然后如第三个代码片段中展示的那样，bar.js 中的程序会加载或导入这两个模块并使用它们。

`import` 可以将一个模块中的一个或多个 API 导入到当前作用域中，并分别绑定在一个变量上（在我们的例子里是 `hello`）。`module` 会将整个模块的 API 导入并绑定到一个变量上（在我们的例子里是 `foo` 和 `bar`）。`export` 会将当前模块的一个标识符（变量、函数）导出为公共 API。这些操作可以在模块定义中根据需要使用任意多次。

模块文件中的内容会被当作好像包含在作用域闭包中一样来处理，就和前面介绍的函数闭包模块一样。

## 5.6 小结

闭包就好像从 JavaScript 中分离出来的一个充满神秘色彩的未开化世界，只有最勇敢的人才能够到达那里。但实际上它只是一个标准，显然就是关于如何在函数作为值按需传递的词法环境中书写代码的。

*当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这时就产生了闭包。*

如果没能认出闭包，也不了解它的工作原理，在使用它的过程中就很容易犯错，比如在循环中。但同时闭包也是一个非常强大的工具，可以用多种形式来实现模块等模式。

模块有两个主要特征：（1）为创建内部作用域而调用了一个包装函数；（2）包装函数的返回值必须至少包括一个对内部函数的引用，这样就会创建涵盖整个包装函数内部作用域的闭包。

现在我们会发现代码中到处都有闭包存在，并且我们能够识别闭包然后用它来做一些有用的事！

附录A

# 动态作用域

在第 2 章中，我们对比了动态作用域和词法作用域模型，JavaScript 中的作用域就是词法作用域（事实上大部分语言都是基于词法作用域的）。

我们会简要地分析一下动态作用域，重申它与词法作用域的区别。但实际上动态作用域是 JavaScript 另一个重要机制 this 的表亲，本书第二部分“this 和对象原型”中会有详细介绍。

从第 2 章中可知，词法作用域是一套关于引擎如何寻找变量以及会在何处找到变量的规则。词法作用域最重要的特征是它的定义过程发生在代码的书写阶段（假设你没有使用 eval() 或 with）。

动态作用域似乎暗示有很好的理由让作用域作为一个在运行时就被动态确定的形式，而不是在写代码时进行静态确定的形式，事实上也是这样的。我们通过示例代码来说明：

```
function foo() {
    console.log( a ); // 2
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

词法作用域让 foo() 中的 a 通过 RHS 引用到了全局作用域中的 a，因此会输出 2。

而动态作用域并不关心函数和作用域是如何声明以及在何处声明的，只关心它们从何处调用。换句话说，作用域链是基于调用栈的，而不是代码中的作用域嵌套。

因此，如果 JavaScript 具有动态作用域，理论上，下面代码中的 foo() 在执行时将会输出 3。

```
function foo() {
    console.log( a ); // 3（不是 2！）
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

为什么会这样？因为当 foo() 无法找到 a 的变量引用时，会顺着调用栈在调用 foo() 的地方查找 a，而不是在嵌套的词法作用域链中向上查找。由于 foo() 是在 bar() 中调用的，引擎会检查 bar() 的作用域，并在其中找到值为 3 的变量 a。

很奇怪吧？现在你可能会这么想。

但这其实是因为你可能只写过基于词法作用域的代码（或者至少以词法作用域为基础进行了深入的思考），因此对动态作用域感到陌生。如果你只用基于动态作用域的语言写过代码，就会觉得这是很自然的，而词法作用域看上去才怪怪的。

需要明确的是，事实上 JavaScript 并不具有动态作用域。它只有词法作用域，简单明了。但是 this 机制某种程度上很像动态作用域。

主要区别：词法作用域是在写代码或者说定义时确定的，而动态作用域是在运行时确定的。（this 也是！）词法作用域关注函数在何处声明，而动态作用域关注函数从何处调用。

最后，this 关注函数如何调用，这就表明了 this 机制和动态作用域之间的关系多么紧密。如果想了解更多关于 this 的详细内容，参见本书第二部分“this 和对象原型”。

# 附录B

# 块作用域的替代方案

第 3 章深入研究了块作用域。至少从 ES3 发布以来，JavaScript 中就有了块作用域，而 `with` 和 `catch` 分句就是块作用域的两个小例子。

但随着 ES6 中引入了 `let`，我们的代码终于有了创建完整、不受约束的块作用域的能力。块作用域在功能上和代码风格上都拥有很多激动人心的新特性。

但如果我们想在 ES6 之前的环境中使用块作用域呢？

考虑下面的代码：

```
{
    let a = 2;
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

这段代码在 ES6 环境中可以正常工作。但是在 ES6 之前的环境中如何才能实现这个效果？答案是使用 `catch`。

```
try{throw 2;}catch(a){
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

天啊！这些代码既丑陋又奇怪。我们看见一个会强制抛出错误的 `try/catch`，但是它抛出

的错误就是一个值 2，然后 catch 分句中的变量声明会接收到这个值。头疼！

没错，catch 分句具有块作用域，因此它可以在 ES6 之前的环境中作为块作用域的替代方案。

“但是，”你可能会说，“鬼才要写这么丑陋的代码！”没错，没人写的代码像 CoffeeScript 编译器输出的代码，但这不是重点。

重点是工具可以将 ES6 的代码转换成能在 ES6 之前环境中运行的形式。你可以使用块作用域来写代码，并享受它带来的好处，然后在构建时通过工具来对代码进行预处理，使之可以在部署时正常工作。

事实上，这是向 ES6 中的所有（好吧，不是所有而是大部分）功能迁移的首选方式：在从 ES6 之前的环境向 ES6 过渡时，使用代码转换工具来对 ES6 代码进行处理，生成兼容 ES5 的代码。

# B.1　Traceur

Google 维护着一个名为 Traceur 的项目，该项目正是用来将 ES6 代码转换成兼容 ES6 之前的环境（大部分是 ES5，但不是全部）。TC39 委员会依赖这个工具（也有其他工具）来测试他们指定的语义化相关的功能。

Traceur 会将我们的代码片段转换成什么样子？你能猜到的！

```
{
    try {
        throw undefined;
    } catch (a) {
        a = 2;
        console.log( a );
    }
}

console.log( a );
```

通过使用这样的工具，我们就可以在使用块作用域时无需考虑目标平台是否是 ES6 环境，因为 try/catch 从 ES3 开始就存在了（并且一直是这样工作的）。

# B.2　隐式和显式作用域

在第 3 章中介绍块作用域时，我们的代码有一些可维护性和可扩展性方面的缺陷。有没有其他可以使用块作用域，并且还能避免这种缺陷的途径？

考虑下面这种 let 的使用方法，它被称作 let 作用域或 let 声明（对比前面的 let 定义）。

```
let (a = 2) {
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

同隐式地劫持一个已经存在的作用域不同，`let` 声明会创建一个显示的作用域并与其进行绑定。显式作用域不仅更加突出，在代码重构时也表现得更加健壮。在语法上，通过强制性地将所有变量声明提升到块的顶部来产生更简洁的代码。这样更容易判断变量是否属于某个作用域。

这种模式同很多人在函数作用域中手动将 `var` 声明提升到函数顶部的方式很接近。`let` 声明有意将变量声明放在块的顶部，如果你并没有到处使用 `let` 定义，那么你的块作用域就很容易辨识和维护。

但是这里有一个小问题，`let` 声明并不包含在 ES6 中。官方的 Traceur 编译器也不接受这种形式的代码。

我们有两个选择，使用合法的 ES6 语法并且在代码规范性上做一些妥协。

```
/*let*/ { let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

工具就是用来解决问题的。因此另外一个选择就是编写显式 `let` 声明，然后通过工具将其转换成合法的、可以工作的代码。

因此我开发了一个名为 let-er 的工具来解决这个问题。let-er 是一个构建时的代码转换器，但它唯一的作用就是找到 `let` 声明并对其进行转换。它不会处理包括 `let` 定义在内的任何其他代码。你可以安全地将 let-er 应用在 ES6 代码转换的第一步，如果有必要，接下来也可以把代码传递给 Traceur 等工具。

此外，let-er 还有一个设置项 `--es6`，开启它（默认是关闭的）会改变生成代码的种类。开启这个设置项时 let-er 会生成完全标准的 ES6 代码，而不会生成通过 `try/catch` 进行 hack 的 ES3 替代方案：

```
{
    let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

因此你马上就可以在 ES6 之前的所有环境中使用 let-er，当你只关注 ES6 环境时，可以开

启设置项，这样就会生成标准的 ES6 代码。

更重要的，你甚至可以使用尚未成为 ES 官方标准的、更加好用的显式 `let` 声明。

## B.3 性能

最后简单地看一下 `try/catch` 带来的性能问题，并尝试回答“为什么不直接使用 IIFE 来创建作用域”这个问题。

首先，`try/catch` 的性能的确很糟糕，但技术层面上没有合理的理由来说明 `try/catch` 必须这么慢，或者会一直慢下去。自从 TC39 支持在 ES6 的转换器中使用 `try/catch` 后，Traceur 团队已经要求 Chrome 对 `try/catch` 的性能进行改进，他们显然有很充分的动机来做这件事情。

其次，IIFE 和 `try/catch` 并不是完全等价的，因为如果将一段代码中的任意一部分拿出来用函数进行包裹，会改变这段代码的含义，其中的 `this`、`return`、`break` 和 `contine` 都会发生变化。IIFE 并不是一个普适的解决方案，它只适合在某些情况下进行手动操作。

最后问题就变成了：你是否想要块作用域？如果你想要，这些工具就可以帮助你。如果不想要，继续使用 `var` 来写代码就好了！

## 附录C

# this词法

尽管这个标题没有详细说明 this 机制，但是 ES6 中有一个主题用非常重要的方式将 this 同词法作用域联系起来了，我们会简单地讨论一下。

ES6 添加了一个特殊的语法形式用于函数声明，叫作*箭头函数*。它看起来是下面这样的：

```
var foo = a => {
    console.log( a );
};

foo( 2 ); // 2
```

这里称作“胖箭头”的写法通常被当作*单调乏味且冗长*（挖苦）的 function 关键字的简写。

但是箭头函数除了让你在声明函数时少敲几次键盘以外，还有更重要的作用。简单来说，下面的代码有问题：

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
        console.log( this.id );
    }
};

var id = "not awesome"

obj.cool(); // 酷

setTimeout( obj.cool, 100 ); // 不酷
```

问题在于 cool() 函数丢失了同 this 之间的绑定。解决这个问题有好几种办法，但最长用的就是 var self = this;。

使用起来如下所示：

```
var obj = {
    count: 0,
    cool: function coolFn() {
        var self = this;

        if (self.count < 1) {
            setTimeout( function timer(){
                self.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // 酷吧?
```

var self = this 这种解决方案圆满解决了理解和正确使用 this 绑定的问题，并且没有把问题过于复杂化，它使用的是我们非常熟悉的工具：词法作用域。self 只是一个可以通过词法作用域和闭包进行引用的标识符，不关心 this 绑定的过程中发生了什么。

人们不喜欢写冗长的东西，尤其是一遍又一遍地写。因此 ES6 的一个初衷就是帮助人们减少重复的场景，事实上包括修复某些习惯用法的问题，this 就是其中一个。

ES6 中的箭头函数引入了一个叫作 this 词法的行为：

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( () => { // 箭头函数是什么鬼东西?
                this.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // 很酷吧 ?
```

简单来说，箭头函数在涉及 this 绑定时的行为和普通函数的行为完全不一致。它放弃了所有普通 this 绑定的规则，取而代之的是用当前的词法作用域覆盖了 this 本来的值。

因此，这个代码片段中的箭头函数并非是以某种不可预测的方式同所属的 this 进行了解绑定，而只是“继承”了 cool() 函数的 this 绑定（因此调用它并不会出错）。

这样除了可以少写一些代码，我认为箭头函数将程序员们经常犯的一个错误给标准化了，也就是混淆了 this 绑定规则和词法作用域规则。

换句话说：为什么要自找麻烦使用 this 风格的代码模式呢？把它和词法作用域结合在一起非常让人头疼。在代码中使用两种风格其中的一种是非常自然的事情，但是不要将两种风格混在一起使用。

> 另一个导致箭头函数不够理想的原因是它们是匿名而非具名的。具名函数比匿名函数更可取的原因参见第 3 章。

在我看来，解决这个“问题”的另一个更合适的办法是正确使用和包含 this 机制。

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( function timer(){
                this.count++; // this 是安全的
                              // 因为 bind(..)
                console.log( "more awesome" );
            }.bind( this ), 100 ); // look, bind()!
        }
    }
 };

obj.cool(); // 更酷了。
```

无论你是喜欢箭头函数中 this 词法的新行为模式，还是喜欢更靠得住的 bind()，都需要注意箭头函数不仅仅意味着可以少写代码。

它们之间有意为之的不同行为需要我们理解和掌握，才能正确地使用它们。

现在我们已经完全理解了词法作用域（还有闭包），理解 this 词法是小菜一碟！

# 附录D

# 致谢

对于本书和这个系列来说，我有许多想要感谢的人。

首先，我必须感谢我的妻子 Christen Simpson，还要感谢我的两个孩子 Ethan 和 Emily 忍受他们的父亲一直摆弄电脑。即使没有在写书，对于 JavaScript 的痴迷也让我的眼睛一刻不离屏幕，我太不应该这样做了。本书之所以可以非常深入和完整地解释 JavaScript，完全是因为我牺牲了陪伴家人的时间。我亏欠家人的实在太多。

我还要感谢我在 O'Reilly 的编辑 Simon St.Laurent 和 Brian MacDonaldm，以及全体编辑和营销人员。与他们共事真是太棒了，他们在“开源”图书写作、编辑和出版的这次尝试过程中对我无比包容。

感谢在该系列书编写过程提出修改建议和帮忙纠错的朋友，他们的贡献令本书增色不少，这些人包括但不限于 Shelley Powers、Tim Ferro、Evan Borden、Forrest L Norvell、Jennifer Davis、Jesse Harlin。感谢 Shane Hudson 为本书第一部分“作用域和闭包”作序，写得很棒。

感谢社区中的所有人，包括 TC39 委员会的成员，他们分享了许多知识并且一直耐心而且细致地回答我接连不断的问题。这些人包括但不限于 John-David Dalton、Juriy“kangax”Zaytsev、Mathias Bynens、Rick Waldron、Axel Rauschmayer、Nicholas Zakas、Angus Croll、Jordan Harband、Reginald Braithwaite、Dave Herman、Brendan Eich、Allen Wirfs-Brock、Bradley Meck、Domenic Denicola、David Walsh、Tim Disney、Kris Kowal、Peter van der Zee、Andrea Giammarchi、Kit Cambridge 以及其他人，原谅我无法在这里一一列出。

由于本系列脱胎于 Kickstarter，我同样想感谢所有（接近）500 个支持者，没有他们就没有这个系列：Jan Szpila、nokiko、Murali Krishnamoorthy、Ryan Joy、Craig Patchett、pdqtrader、Dale Fukami、ray hatfield、R0drigo Perez [Mx]、Dan Petitt、Jack Franklin、Andrew Berry、Brian Grinstead、Rob Sutherland、Sergi Meseguer、Phillip Gourley、Mark Watson、Jeff Carouth、Alfredo Sumaran、Martin Sachse、Marcio Barrios、Dan、AimelyneM、Matt Sullivan、Delnatte Pierre-Antoine、Jake Smith、Eugen Tudorancea、Iris、David Trinh、simonstl、Ray Daly、Uros Gruber、Justin Myers、Shai Zonis、Mom & Dad、Devin Clark、Dennis Palmer、Brian Panahi Johnson、Josh Marshall、Marshall、Dennis Kerr、Matt Steele、Erik Slagter、Sacah、Justin Rainbow、Christian Nilsson、Delapouite、D. Pereira、Nicolas Hoizey、George V. Reilly、Dan Reeves、Bruno Laturner、Chad Jennings、Shane King、Jeremiah Lee Cohick、od3n、Stan Yamane、Marko Vucinic、Jim B、Stephen Collins、Ægir Þorsteinsson、Eric Pederson、Owain、Nathan Smith、Jeanetteurphy、Alexandre ELISÉ、Chris Peterson、Rik Watson、Luke Matthews、Justin Lowery、Morten Nielsen、Vernon Kesner、Chetan Shenoy、Paul Tregoing、Marc Grabanski、Dion Almaer、Andrew Sullivan、Keith Elsass、Tom Burke、Brian Ashenfelter、David Stuart、Karl Swedberg、Graeme、Brandon Hays、John Christopher、Gior、manoj reddy、Chad Smith、Jared Harbour、Minoru TODA、Chris Wigley、Daniel Mee、Mike、Handyface、Alex Jahraus、Carl Furrow、Rob Foulkrod、Max Shishkin、Leigh Penny Jr.、Robert Ferguson、Mike van Hoenselaar、Hasse Schougaard、rajan venkataguru、Jeff Adams、Trae Robbins、Rolf Langenhuijzen、Jorge Antunes、Alex Koloskov、Hugh Greenish、Tim Jones、Jose Ochoa、Michael Brennan-White、Naga Harish Muvva、Barkóczi Dávid、Kitt Hodsden、Paul McGraw、Sascha Goldhofer、Andrew Metcalf、Markus Krogh、Michael Mathews、Matt Jared、Juanfran、Georgie Kirschner、Kenny Lee、Ted Zhang、Amit Pahwa、Inbal Sinai、Dan Raine、Schabse Laks、Michael Tervoort、Alexandre Abreu、Alan Joseph Williams、NicolasD、Cindy Wong、Reg Braithwaite、LocalPCGuy、Jon Friskics、Chris Merriman、John Pena、Jacob Katz、Sue Lockwood、Magnus Johansson、Jeremy Crapsey、Grzegorz Pawłowski、nico nuzzaci、Christine Wilks、Hans Bergren、charles montgomery、Ariel בר-לבב Fogel、Ivan Kolev、Daniel Campos、Hugh Wood、Christian Bradford、Frédéric Harper、Ionuț Dan Popa、Jeff Trimble、Rupert Wood、Trey Carrico、Pancho Lopez、Joël kuijten、Tom A Marra、Jeff Jewiss、Jacob Rios、Paolo Di Stefano、Soledad Penades、Chris Gerber、Andrey Dolganov、Wil MooreIII、Thomas Martineau、Kareem、Ben Thouret、Udi Nir、Morgan Laupies、jory carson-burson、Nathan L Smith、Eric Damon Walters、Derry Lozano-Hoyland、Geoffrey Wiseman、mkeehner、KatieK、Scott MacFarlane、Brian LaShomb、Adrien Mas、christopher ross、Ian Littman、Dan Atkinson、Elliot Jobe、Nick Dozier、Peter Wooley、John Hoover、dan、Martin A. Jackson、Héctor Fernando Hurtado、andy ennamorato、Paul Seltmann、Melissa Gore、Dave Pollard、Jack Smith、Philip Da Silva、Guy Israeli、@megalithic、Damian Crawford、Felix Gliesche、April Carter Grant、Heidi、

jim tierney、Andrea Giammarchi、Nico Vignola、Don Jones、Chris Hartjes、Alex Howes、john gibbon、David J. Groom、BBox、Yu *Dilys* Sun、Nate Steiner、Brandon Satrom、Brian Wyant、Wesley Hales、Ian Pouncey、Timothy Kevin Oxley、George Terezakis、sanjay raj、Jordan Harband、Marko McLion、Wolfgang Kaufmann、Pascal Peuckert、Dave Nugent、Markus Liebelt、Welling Guzman、Nick Cooley、Daniel Mesquita、Robert Syvarth、Chris Coyier、Rémy Bach、Adam Dougal、Alistair Duggin、David Loidolt、Ed Richer、Brian Chenault、GoldFire Studios、Carles Andrés、Carlos Cabo、Yuya Saito、roberto ricardo、Barnett Klane、Mike Moore、Kevin Marx、Justin Love、Joe Taylor、Paul Dijou、Michael Kohler、Rob Cassie、Mike Tierney、Cody Leroy Lindley、tofuji、Shimon Schwartz、Raymond、Luc De Brouwer、David Hayes、Rhys Brett-Bowen、Dmitry、Aziz Khoury、Dean、Scott Tolinski-Level Up、Clement Boirie、Djordje Lukic、Anton Kotenko、Rafael Corral、Philip Hurwitz、Jonathan Pidgeon、Jason Campbell、Joseph C.、SwiftOne、Jan Hohner、Derick Bailey、getify、Daniel Cousineau、Chris Charlton、Eric Turner、David Turner、Joël Galeran、Dharma Vagabond、adam、Dirk van Bergen、dave ♥♫★ furf、Vedran Zakanj、Ryan McAllen、Natalie Patrice Tucker、Eric J. Bivona、Adam Spooner、Aaron Cavano、Kelly Packer、Eric J、Martin Drenovac、Emilis、Michael Pelikan、Scott F. Walter、Josh Freeman、Brandon Hudgeons、vijay chennupati、Bill Glennon、Robin R.、Troy Forster、otaku_coder、Brad、Scott、Frederick Ostrander、Adam Brill、Seb Flippence、Michael Anderson、Jacob、Adam Randlett、Standard、Joshua Clanton、Sebastian Kouba、Chris Deck、SwordFire、Hannes Papenberg、Richard Woeber、hnzz、Rob Crowther、Jedidiah Broadbent、Sergey Chernyshev、Jay-Ar Jamon、Ben Combee、luciano bonachela、Mark Tomlinson、Kit Cambridge、Michael Melgares、Jacob Adams、Adrian Bruinhout、Bev Wieber、Scott Puleo、Thomas Herzog、April Leone、Daniel Mizieliński、Kees van Ginkel、Jon Abrams、Erwin Heiser、Avi Laviad、David newell、Jean- Francois Turcot、Niko Roberts、Erik Dana、Charles Neill、Aaron Holmes、Grzegorz Ziółkowski、Nathan Youngman、Timothy、Jacob Mather、Michael Allan、Mohit Seth、Ryan Ewing、Benjamin Van Treese,Marcelo Santos、Denis Wolf、Phil Keys、Chris Yung、Timo Tijhof、Martin Lekvall、Agendine、Greg Whitworth、Helen Humphrey、Dougal Campbell、Johannes Harth、Bruno Girin、Brian Hough、Darren Newton、Craig McPheat、Olivier Tille、Dennis Roethig、Mathias Bynens、Brendan Stromberger、sundeep、John Meyer、Ron Male、John F Croston III、gigante、Carl Bergenhem、B.J. May、Rebekah Tyler、Ted Foxberry、Jordan Reese、Terry Suitor、afeliz、Tom Kiefer、Darragh Duffy、Kevin Vanderbeken、Andy Pearson、Simon Mac Donald、Abid Din、Chris Joel、Tomas Theunissen、David Dick、Paul Grock、Brandon Wood、John Weis、dgrebb、Nick Jenkins、Chuck Lane、Johnny Megahan、marzsman、Tatu Tamminen、Geoffrey Knauth、Alexander Tarmolov、Jeremy Tymes、Chad Auld、Sean Parmelee、Rob Staenke、Dan Bender、Yannick derwa、Joshua Jones、Geert Plaisier、Tom LeZotte、Christen Simpson、Stefan Bruvik、Justin Falcone、Carlos Santana、Michael Weiss、Pablo Villoslada、

Peter deHaan、Dimitris Iliopoulos、seyDoggy、Adam Jordens、Noah Kantrowitz、Amol M、Matthew Winnard、Dirk Ginader、Phinam Bui、David Rapson、Andrew Baxter、Florian Bougel、Michael George、Alban Escalier、Daniel Sellers、Sasha Rudan、John Green、Robert Kowalski、David I. Teixeira (@ditma、Charles Carpenter、Justin Yost、Sam S、Denis Ciccale、Kevin Sheurs、Yannick Croissant、Pau Fracés、Stephen McGowan、Shawn Searcy、Chris Ruppel、Kevin Lamping、Jessica Campbell、Christopher Schmitt、Sablons、Jonathan Reisdorf、Bunni Gek、Teddy Huff、Michael Mullany、Michael Fürstenberg、Carl Henderson、Rick Yoesting、Scott Nichols、Hernán Ciudad、Andrew Maier、Mike Stapp、Jesse Shawl、Sérgio Lopes、jsulak、Shawn Price、Joel Clermont、Chris Ridmann、Sean Timm、Jason Finch、Aiden Montgomery、Elijah Manor、Derek Gathright、Jesse Harlin、Dillon Curry、Courtney Myers、Diego Cadenas、Arne de Bree、João Paulo Dubas、James Taylor、Philipp Kraeutli、Mihai Paun、Sam Gharegozlou、joshjs、Matt Murchison、Eric Windham、Timo Behrmann、Andrew Hall、joshua price、and Théophile Villard。

本系列书的写作是开源的，包括编辑和出版。我们要感谢 GitHub 提供的社区协作功能！

再次感谢，我无法列出所有人的名字，但是非常感谢你们。希望本书可以被所有人“拥有”，能促进大家对于 JavaScript 语言的理解，能帮助现在和未来的社区贡献者们。

# 第二部分

# this和对象原型

［美］Kyle Simpson 著
梁杰 译

# 序

读这本书准备作序的时候，我不禁想起 15 年前学习 JavaScript 时的情景。过去的这 15 年中，我一直用它进行编程和开发，同时，JavaScript 也在不断发生变化。

15 年前我刚开始使用 JavaScript 时，在网页中使用 CSS 和 JavaScript 等非 HTML 技术被称为 DHTML 或者动态 HTML。在那之后，JavaScript 的用途发生了巨大的变化，印象中主要用于给网页添加动态雪花或者在状态栏中添加动态时钟。说实话，职业生涯的早期我并没有对 JavaScript 给予足够的重视，因为在我看来它主要的功能就是编写一些有趣的小东西。

直到 2005 年我才第一次认识到 JavaScript 是一门真正的编程语言，应当受到我的重视。仔细研究了谷歌地图的第一个测试版本之后，我被它的潜力深深地吸引住了。在那时，谷歌地图是一个史无前例的应用——你可以用鼠标移动和缩放地图，并且可以在不重载页面的情况下发起服务器请求——这些全部用 JavaScript 完成，简直就像魔法一样！

如果某些事情对你来说像魔法一样，那意味着你看到了新生事物的曙光。我的想法是正确的——今天，无论在客户端还是服务端，JavaScript 都已经成为了我的一门主要编程语言，没有其他语言比它更适合完成这些工作。

回顾这 15 年，有一件事我很后悔，那就是没有在 2005 年之前给予 JavaScript 足够的重视。更准确地说，我并没有想到 JavaScript 会像 C++、C#、Java 等语言一样，成为一门非常有用的真正的编程语言。

如果在一开始时就能遇到“你不知道的 JavaScript”系列丛书，我的整个职业生涯都会大不相同。对于这个系列丛书，我非常欣赏的一点是：它可以用有趣并且有效的方式帮助你构建起对于 JavaScript 的理解。

本书第二部分“this 和对象原型”非常不错，它很好地衔接了本书第一部分“作用域和闭包”，进一步介绍了 JavaScript 语言中非常重要的两个部分，this 关键字和原型。这两个部分对于你未来的学习来说非常重要，它们是使用 JavaScript 进行编程的基础。只有掌握了如何创建、关联和扩展对象，你才能用 JavaScript 创建类似谷歌地图这样大型的复杂应用。

在我看来，绝大多数 Web 开发者可能都没有创建过一个 JavaScript 对象，他们只是把 JavaScript 当作按钮和 AJAX 请求之间的事件绑定粘合剂。其实我也曾经是他们中的一员，但是当我学会在 JavaScript 中使用原型和创建对象之后，整个世界都变样了。如果你也只会编写事件绑定代码，那么这本书是非读不可的；如果你只想复习一下的话，这本书也一定是首选。无论如何，你绝对不会失望的，相信我！

——Nick Berardi
nickberardi.com
Twitter：@nberardi

# 第 1 章

# 关于this

this 关键字是 JavaScript 中最复杂的机制之一。它是一个很特别的关键字，被自动定义在所有函数的作用域中。但是即使是非常有经验的 JavaScript 开发者也很难说清它到底指向什么。

> 任何足够先进的技术都和魔法无异。
>
> ——Arthur C. Clarke

实际上，JavaScript 中 this 的机制并没有那么先进，但是开发者往往会把理解过程复杂化，毫无疑问，在缺乏清晰认识的情况下，this 对你来说完全就是一种魔法。

> "this"是沟通过程中极其常见的一个代词。所以，在交流过程中很难区分我们到底把"this"当作代词还是当作关键字。清晰起见，我总一直使用 this 表示关键字，使用"this"或者 this 来表示代词。

## 1.1 为什么要用this

如果对于有经验的 JavaScript 开发者来说 this 都是一种非常复杂的机制，那它到底有用在哪里呢？真的值得我们付出这么大的代价学习吗？的确，在介绍怎么做之前我们需要先明白为什么。

下面我们来解释一下为什么要使用 this：

```
function identify() {
    return this.name.toUpperCase();
}

function speak() {
    var greeting = "Hello, I'm " + identify.call( this );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER

speak.call( me ); // Hello，我是 KYLE
speak.call( you ); // Hello，我是 READER
```

看不懂这段代码？不用担心！我们很快就会讲解。现在请暂时抛开这些问题，专注于为什么。

这段代码可以在不同的上下文对象（me 和 you）中重复使用函数 identify() 和 speak()，不用针对每个对象编写不同版本的函数。

如果不使用 this，那就需要给 identify() 和 speak() 显式传入一个上下文对象。

```
function identify(context) {
    return context.name.toUpperCase();
}

function speak(context) {
    var greeting = "Hello, I'm " + identify( context );
    console.log( greeting );
}

identify( you ); // READER
speak( me ); //hello，我是 KYLE
```

然而，this 提供了一种更优雅的方式来隐式"传递"一个对象引用，因此可以将 API 设计得更加简洁并且易于复用。

随着你的使用模式越来越复杂，显式传递上下文对象会让代码变得越来越混乱，使用 this 则不会这样。当我们介绍对象和原型时，你就会明白函数可以自动引用合适的上下文对象有多重要。

## 1.2 误解

我们之后会解释 this 到底是如何工作的，但是首先需要消除一些关于 this 的错误认识。

太拘泥于“this”的字面意思就会产生一些误解。有两种常见的对于 this 的解释，但是它们都是错误的。

### 1.2.1 指向自身

人们很容易把 this 理解成指向函数自身，这个推断从英语的语法角度来说是说得通的。

那么为什么需要从函数内部引用函数自身呢？常见的原因是递归（从函数内部调用这个函数）或者可以写一个在第一次被调用后自己解除绑定的事件处理器。

JavaScript 的新手开发者通常会认为，既然函数看作一个对象（JavaScript 中的所有函数都是对象），那就可以在调用函数时存储状态（属性的值）。这是可行的，有些时候也确实有用，但是在本书即将介绍的许多模式中你会发现，除了函数对象还有许多更合适存储状态的地方。

不过现在我们先来分析一下这个模式，让大家看到 this 并不像我们所想的那样指向函数本身。

我们想要记录一下函数 foo 被调用的次数，思考一下下面的代码：

```
function foo(num) {
    console.log( "foo: " + num );

    // 记录 foo 被调用的次数
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo 被调用了多少次？
console.log( foo.count ); // 0 -- WTF?
```

console.log 语句产生了 4 条输出，证明 foo(..) 确实被调用了 4 次，但是 foo.count 仍然是 0。显然从字面意思来理解 this 是错误的。

执行 foo.count = 0 时，的确向函数对象 foo 添加了一个属性 count。但是函数内部代码 this.count 中的 this 并不是指向那个函数对象，所以虽然属性名相同，根对象却并不相同，困惑随之产生。

> 负责的开发者一定会问“如果我增加的 count 属性和预期的不一样，那我增加的是哪个 count ？”实际上，如果他深入探索的话，就会发现这段代码在无意中创建了一个全局变量 count（原理参见第 2 章），它的值为 NaN。当然，如果他发现了这个奇怪的结果，那一定会接着问：“为什么它是全局的，为什么它的值是 NaN 而不是其他更合适的值？”（参见第 2 章。）

遇到这样的问题时，许多开发者并不会深入思考为什么 this 的行为和预期的不一致，也不会试图回答那些很难解决但却非常重要的问题。他们只会回避这个问题并使用其他方法来达到目的，比如创建另一个带有 count 属性的对象。

```
function foo(num) {
    console.log( "foo: " + num );

    // 记录 foo 被调用的次数
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo 被调用了多少次？
console.log( data.count ); // 4
```

从某种角度来说这个方法确实“解决”了问题，但可惜它忽略了真正的问题——无法理解 this 的含义和工作原理——而是返回舒适区，使用了一种更熟悉的技术：词法作用域。

> 词法作用域是一种非常优秀并且有用的技术。我丝毫没有贬低它的意思（可以参考本书第一部分“作用域和闭包”）。但是如果你仅仅是因为无法猜对 this 的用法，就放弃学习 this 而去使用词法作用域，就不能算是一种很好的解决办法了。

如果要从函数对象内部引用它自身，那只使用 this 是不够的。一般来说你需要通过一个指向函数对象的词法标识符（变量）来引用它。

思考一下下面这两个函数：

```
function foo() {
    foo.count = 4; // foo 指向它自身
}

setTimeout( function(){
    // 匿名（没有名字的）函数无法指向自身
}, 10 );
```

第一个函数被称为具名函数，在它内部可以使用 foo 来引用自身。

但是在第二个例子中，传入 setTimeout(..) 的回调函数没有名称标识符（这种函数被称为匿名函数），因此无法从函数内部引用自身。

> 还有一种传统的但是现在已经被弃用和批判的用法，是使用 arguments.callee 来引用当前正在运行的函数对象。这是唯一一种可以从匿名函数对象内部引用自身的方法。然而，更好的方式是避免使用匿名函数，至少在需要自引用时使用具名函数（表达式）。arguments.callee 已经被弃用，不应该再使用它。

所以，对于我们的例子来说，另一种解决方法是使用 foo 标识符替代 this 来引用函数对象：

```
function foo(num) {
    console.log( "foo: " + num );

    // 记录 foo 被调用的次数
    foo.count++;
}
foo.count=0
var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
```

```
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo 被调用了多少次?
console.log( foo.count ); // 4
```

然而，这种方法同样回避了 this 的问题，并且完全依赖于变量 foo 的词法作用域。

另一种方法是强制 this 指向 foo 函数对象：

```
function foo(num) {
    console.log( "foo: " + num );

    // 记录 foo 被调用的次数
    // 注意，在当前的调用方式下（参见下方代码），this 确实指向 foo
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        // 使用 call(..) 可以确保 this 指向函数对象 foo 本身
        foo.call( foo, i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo 被调用了多少次?
console.log( foo.count ); // 4
```

这次我们接受了 this，没有回避它。如果你仍然感到困惑的话，不用担心，之后我们会详细解释具体的原理。

### 1.2.2　它的作用域

第二种常见的误解是，this 指向函数的作用域。这个问题有点复杂，因为在某种情况下它是正确的，但是在其他情况下它却是错误的。

需要明确的是，this 在任何情况下都不指向函数的词法作用域。在 JavaScript 内部，作用域确实和对象类似，可见的标识符都是它的属性。但是作用域“对象”无法通过 JavaScript 代码访问，它存在于 JavaScript 引擎内部。

思考一下下面的代码，它试图（但是没有成功）跨越边界，使用 this 来隐式引用函数的词法作用域：

```
function foo() {
    var a = 2;
    this.bar();
}

function bar() {
    console.log( this.a );
}

foo(); // ReferenceError: a is not defined
```

这段代码中的错误不止一个。虽然这段代码看起来好像是我们故意写出来的例子，但是实际上它出自一个公共社区中互助论坛的精华代码。这段代码非常完美（同时也令人伤感）地展示了 this 多么容易误导人。

首先，这段代码试图通过 this.bar() 来引用 bar() 函数。这是绝对不可能成功的，我们之后会解释原因。调用 bar() 最自然的方法是省略前面的 this，直接使用词法引用标识符。

此外，编写这段代码的开发者还试图使用 this 联通 foo() 和 bar() 的词法作用域，从而让 bar() 可以访问 foo() 作用域里的变量 a。这是不可能实现的，你不能使用 this 来引用一个词法作用域内部的东西。

每当你想要把 this 和词法作用域的查找混合使用时，一定要提醒自己，这是无法实现的。

## 1.3 this到底是什么

排除了一些错误理解之后，我们来看看 this 到底是一种什么样的机制。

之前我们说过 this 是在运行时进行绑定的，并不是在编写时绑定，它的上下文取决于函数调用时的各种条件。this 的绑定和函数声明的位置没有任何关系，只取决于函数的调用方式。

当一个函数被调用时，会创建一个活动记录（有时候也称为执行上下文）。这个记录会包含函数在哪里被调用（调用栈）、函数的调用方法、传入的参数等信息。this 就是记录的其中一个属性，会在函数执行的过程中用到。

在下一章我们会学习如何寻找函数的调用位置，从而判断函数在执行过程中会如何绑定 this。

## 1.4 小结

对于那些没有投入时间学习 this 机制的 JavaScript 开发者来说，this 的绑定一直是一件非

常令人困惑的事。`this` 是非常重要的，但是猜测、尝试并出错和盲目地从 Stack Overflow 上复制和粘贴答案并不能让你真正理解 `this` 的机制。

学习 `this` 的第一步是明白 `this` 既不指向函数自身也不指向函数的词法作用域，你也许被这样的解释误导过，但其实它们都是错误的。

`this` 实际上是在函数被调用时发生的绑定，它指向什么完全取决于函数在哪里被调用。

第 2 章

# this全面解析

在第 1 章中，我们排除了一些对于 this 的错误理解并且明白了每个函数的 this 是在调用时被绑定的，完全取决于函数的调用位置（也就是函数的调用方法）。

## 2.1 调用位置

在理解 this 的绑定过程之前，首先要理解*调用位置*：调用位置就是函数在代码中被调用的位置（而不是声明的位置）。只有仔细分析调用位置才能回答这个问题：这个 this 到底引用的是什么？

通常来说，寻找调用位置就是寻找“函数被调用的位置”，但是做起来并没有这么简单，因为某些编程模式可能会隐藏真正的调用位置。

最重要的是要分析*调用栈*（就是为了到达当前执行位置所调用的所有函数）。我们关心的调用位置就在当前正在执行的函数的前一个调用中。

下面我们来看看到底什么是调用栈和调用位置：

```
function baz() {
    // 当前调用栈是：baz
    // 因此，当前调用位置是全局作用域

    console.log( "baz" );
    bar(); // <-- bar 的调用位置
}

function bar() {
```

```
    // 当前调用栈是 baz -> bar
    // 因此，当前调用位置在 baz 中

    console.log( "bar" );
    foo(); // <-- foo 的调用位置
}

function foo() {
    // 当前调用栈是 baz -> bar -> foo
    // 因此，当前调用位置在 bar 中

    console.log( "foo" );
}

baz(); // <-- baz 的调用位置
```

注意我们是如何（从调用栈中）分析出真正的调用位置的，因为它决定了 this 的绑定。

> 你可以把调用栈想象成一个函数调用链，就像我们在前面代码段的注释中所写的一样。但是这种方法非常麻烦并且容易出错。另一个查看调用栈的方法是使用浏览器的调试工具。绝大多数现代桌面浏览器都内置了开发者工具，其中包含 JavaScript 调试器。就本例来说，你可以在工具中给 foo() 函数的第一行代码设置一个断点，或者直接在第一行代码之前插入一条 debugger; 语句。运行代码时，调试器会在那个位置暂停，同时会展示当前位置的函数调用列表，这就是你的调用栈。因此，如果你想要分析 this 的绑定，使用开发者工具得到调用栈，然后找到栈中第二个元素，这就是真正的调用位置。

## 2.2 绑定规则

我们来看看在函数的执行过程中调用位置如何决定 this 的绑定对象。

你必须找到调用位置，然后判断需要应用下面四条规则中的哪一条。我们首先会分别解释这四条规则，然后解释多条规则都可用时它们的优先级如何排列。

### 2.2.1 默认绑定

首先要介绍的是最常用的函数调用类型：独立函数调用。可以把这条规则看作是无法应用其他规则时的默认规则。

思考一下下面的代码：

```
function foo() {
    console.log( this.a );
}
```

```
var a = 2;

foo(); // 2
```

你应该注意到的第一件事是，声明在全局作用域中的变量（比如 `var a = 2`）就是全局对象的一个同名属性。它们本质上就是同一个东西，并不是通过复制得到的，就像一个硬币的两面一样。

接下来我们可以看到当调用 `foo()` 时，`this.a` 被解析成了全局变量 `a`。为什么？因为在本例中，函数调用时应用了 `this` 的*默认绑定*，因此 `this` 指向全局对象。

那么我们怎么知道这里应用了*默认绑定*呢？可以通过分析调用位置来看看 `foo()` 是如何调用的。在代码中，`foo()` 是直接使用不带任何修饰的函数引用进行调用的，因此只能使用*默认绑定*，无法应用其他规则。

如果使用严格模式（`strict mode`），那么全局对象将无法使用默认绑定，因此 `this` 会绑定到 `undefined`：

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo(); // TypeError: this is undefined
```

这里有一个微妙但是非常重要的细节，虽然 `this` 的绑定规则完全取决于调用位置，但是只有 `foo()` 运行在非 `strict mode` 下时，默认绑定才能绑定到全局对象；严格模式下与 `foo()` 的调用位置无关：

```
function foo() {
    console.log( this.a );
}

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();
```

通常来说你不应该在代码中混合使用 `strict mode` 和 `non-strict mode`。整个程序要么严格要么非严格。然而，有时候你可能会用到第三方库，其严格程度和你的代码有所不同，因此一定要注意这类兼容性细节。

### 2.2.2　隐式绑定

另一条需要考虑的规则是调用位置是否有上下文对象，或者说是否被某个对象拥有或者包含，不过这种说法可能会造成一些误导。

思考下面的代码：

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

首先需要注意的是 `foo()` 的声明方式，及其之后是如何被当作引用属性添加到 `obj` 中的。但是无论是直接在 `obj` 中定义还是先定义再添加为引用属性，这个函数严格来说都不属于 `obj` 对象。

然而，调用位置会*使用* `obj` 上下文来引用函数，因此你可以说函数被调用时 `obj` 对象“拥有”或者“包含”它。

无论你如何称呼这个模式，当 `foo()` 被调用时，它的落脚点确实指向 `obj` 对象。当函数引用有上下文对象时，*隐式绑定*规则会把函数调用中的 `this` 绑定到这个上下文对象。因为调用 `foo()` 时 `this` 被绑定到 `obj`，因此 `this.a` 和 `obj.a` 是一样的。

对象属性引用链中只有最顶层或者说最后一层会影响调用位置。举例来说：

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

#### 隐式丢失

一个最常见的 `this` 绑定问题就是被*隐式绑定*的函数会丢失绑定对象，也就是说它会应用默

认绑定，从而把 this 绑定到全局对象或者 undefined 上，取决于是否是严格模式。

思考下面的代码：

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // 函数别名!

var a = "oops, global"; // a 是全局对象的属性

bar(); // "oops, global"
```

虽然 bar 是 obj.foo 的一个引用，但是实际上，它引用的是 foo 函数本身，因此此时的 bar() 其实是一个不带任何修饰的函数调用，因此应用了默认绑定。

一种更微妙、更常见并且更出乎意料的情况发生在传入回调函数时：

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // fn 其实引用的是 foo

    fn(); // <-- 调用位置!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // a 是全局对象的属性

doFoo( obj.foo ); // "oops, global"
```

参数传递其实就是一种隐式赋值，因此我们传入函数时也会被隐式赋值，所以结果和上一个例子一样。

如果把函数传入语言内置的函数而不是传入你自己声明的函数，会发生什么呢？结果是一样的，没有区别：

```
function foo() {
    console.log( this.a );
```

```
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // a 是全局对象的属性

setTimeout( obj.foo, 100 ); // "oops, global"
```

JavaScript 环境中内置的 `setTimeout()` 函数实现和下面的伪代码类似：

```
function setTimeout(fn,delay) {
    // 等待 delay 毫秒
    fn(); // <-- 调用位置！
}
```

就像我们看到的那样，回调函数丢失 `this` 绑定是非常常见的。除此之外，还有一种情况 `this` 的行为会出乎我们意料：调用回调函数的函数可能会修改 `this`。在一些流行的 JavaScript 库中事件处理器常会把回调函数的 `this` 强制绑定到触发事件的 DOM 元素上。这在一些情况下可能很有用，但是有时它可能会让你感到非常郁闷。遗憾的是，这些工具通常无法选择是否启用这个行为。

无论是哪种情况，`this` 的改变都是意想不到的，实际上你无法控制回调函数的执行方式，因此就没有办法控制会影响绑定的调用位置。之后我们会介绍如何通过固定 `this` 来修复（这里是双关，"修复"和"固定"的英语单词都是 fixing）这个问题。

### 2.2.3 显式绑定

就像我们刚才看到的那样，在分析隐式绑定时，我们必须在一个对象内部包含一个指向函数的属性，并通过这个属性间接引用函数，从而把 `this` 间接（隐式）绑定到这个对象上。

那么如果我们不想在对象内部包含函数引用，而想在某个对象上强制调用函数，该怎么做呢？

JavaScript 中的"所有"函数都有一些有用的特性（这和它们的 [[ 原型 ]] 有关——之后我们会详细介绍原型），可以用来解决这个问题。具体点说，可以使用函数的 `call(..)` 和 `apply(..)` 方法。严格来说，JavaScript 的宿主环境有时会提供一些非常特殊的函数，它们并没有这两个方法。但是这样的函数非常罕见，JavaScript 提供的绝大多数函数以及你自己创建的所有函数都可以使用 `call(..)` 和 `apply(..)` 方法。

这两个方法是如何工作的呢？它们的第一个参数是一个对象，它们会把这个对象绑定到 `this`，接着在调用函数时指定这个 `this`。因为你可以直接指定 `this` 的绑定对象，因此我们称之为显式绑定。

思考下面的代码：

```
function foo() {
    console.log( this.a );
}

var obj = {
    a:2
};

foo.call( obj ); // 2
```

通过 foo.call(..)，我们可以在调用 foo 时强制把它的 this 绑定到 obj 上。

如果你传入了一个原始值（字符串类型、布尔类型或者数字类型）来当作 this 的绑定对象，这个原始值会被转换成它的对象形式（也就是 new String(..)、new Boolean(..) 或者 new Number(..)）。这通常被称为“装箱”。

> 从 this 绑定的角度来说，call(..) 和 apply(..) 是一样的，它们的区别体现在其他的参数上，但是现在我们不用考虑这些。

可惜，*显式绑定*仍然无法解决我们之前提出的丢失绑定问题。

### 1. 硬绑定

但是显式绑定的一个变种可以解决这个问题。

思考下面的代码：

```
function foo() {
    console.log( this.a );
}

var obj = {
    a:2
};

var bar = function() {
    foo.call( obj );
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// 硬绑定的 bar 不可能再修改它的 this
bar.call( window ); // 2
```

我们来看看这个变种到底是怎样工作的。我们创建了函数 bar()，并在它的内部手动调用了 foo.call(obj)，因此强制把 foo 的 this 绑定到了 obj。无论之后如何调用函数 bar，它总会手动在 obj 上调用 foo。这种绑定是一种显式的强制绑定，因此我们称之为*硬绑定*。

硬绑定的典型应用场景就是创建一个包裹函数，传入所有的参数并返回接收到的所有值：

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a:2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

另一种使用方法是创建一个 i 可以重复使用的辅助函数：

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// 简单的辅助绑定函数
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
    };
}

var obj = {
    a:2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

由于*硬绑定*是一种非常常用的模式，所以在 ES5 中提供了内置的方法 Function.prototype.bind，它的用法如下：

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}
```

```
var obj = {
    a:2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

bind(..) 会返回一个硬编码的新函数，它会把参数设置为 this 的上下文并调用原始函数。

#### 2. API调用的“上下文”

第三方库的许多函数，以及 JavaScript 语言和宿主环境中许多新的内置函数，都提供了一个可选的参数，通常被称为“上下文”（context），其作用和 bind(..) 一样，确保你的回调函数使用指定的 this。

举例来说：

```
function foo(el) {
    console.log( el, this.id );
}

var obj = {
    id: "awesome"
};

// 调用 foo(..) 时把 this 绑定到 obj
[1, 2, 3].forEach( foo, obj );
// 1 awesome 2 awesome 3 awesome
```

这些函数实际上就是通过 call(..) 或者 apply(..) 实现了显式绑定，这样你可以少些一些代码。

### 2.2.4　new绑定

这是第四条也是最后一条 this 的绑定规则，在讲解它之前我们首先需要澄清一个非常常见的关于 JavaScript 中函数和对象的误解。

在传统的面向类的语言中，“构造函数”是类中的一些特殊方法，使用 new 初始化类时会调用类中的构造函数。通常的形式是这样的：

```
something = new MyClass(..);
```

JavaScript 也有一个 new 操作符，使用方法看起来也和那些面向类的语言一样，绝大多数开发者都认为 JavaScript 中 new 的机制也和那些语言一样。然而，JavaScript 中 new 的机制实际上和面向类的语言完全不同。

首先我们重新定义一下 JavaScript 中的“构造函数”。在 JavaScript 中，构造函数只是一些使用 new 操作符时被调用的函数。它们并不会属于某个类，也不会实例化一个类。实际上，它们甚至都不能说是一种特殊的函数类型，它们只是被 new 操作符调用的普通函数而已。

举例来说，思考一下 Number(..) 作为构造函数时的行为，ES5.1 中这样描述它：

> 15.7.2　Number 构造函数
> 当 Number 在 new 表达式中被调用时，它是一个构造函数：它会初始化新创建的对象。

所以，包括内置对象函数（比如 Number(..)，详情请查看第 3 章）在内的所有函数都可以用 new 来调用，这种函数调用被称为构造函数调用。这里有一个重要但是非常细微的区别：实际上并不存在所谓的“构造函数”，只有对于函数的“构造调用”。

使用 new 来调用函数，或者说发生构造函数调用时，会自动执行下面的操作。

1. 创建（或者说构造）一个全新的对象。
2. 这个新对象会被执行 [[ 原型 ]] 连接。
3. 这个新对象会绑定到函数调用的 this。
4. 如果函数没有返回其他对象，那么 new 表达式中的函数调用会自动返回这个新对象。

我们现在关心的是第 1 步、第 3 步、第 4 步，所以暂时跳过第 2 步，第 5 章会详细介绍它。

思考下面的代码：

```
function foo(a) {
    this.a = a;
}

var bar = new foo(2);
console.log( bar.a ); // 2
```

使用 new 来调用 foo(..) 时，我们会构造一个新对象并把它绑定到 foo(..) 调用中的 this 上。new 是最后一种可以影响函数调用时 this 绑定行为的方法，我们称之为 new 绑定。

## 2.3　优先级

现在我们已经了解了函数调用中 this 绑定的四条规则，你需要做的就是找到函数的调用位置并判断应当应用哪条规则。但是，如果某个调用位置可以应用多条规则该怎么办？为了解决这个问题就必须给这些规则设定优先级，这就是我们接下来要介绍的内容。

毫无疑问，默认绑定的优先级是四条规则中最低的，所以我们可以先不考虑它。

隐式绑定和显式绑定哪个优先级更高？我们来测试一下：

```
function foo() {
    console.log( this.a );
}

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

可以看到，显式绑定优先级更高，也就是说在判断时应当先考虑是否可以应用显式绑定。

现在我们需要搞清楚 new 绑定和隐式绑定的优先级谁高谁低：

```
function foo(something) {
    this.a = something;
}

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

可以看到 new 绑定比隐式绑定优先级高。但是 new 绑定和显式绑定谁的优先级更高呢？

> new 和 call/apply 无法一起使用，因此无法通过 new foo.call(obj1) 来直接进行测试。但是我们可以使用硬绑定来测试它俩的优先级。

在看代码之前先回忆一下硬绑定是如何工作的。Function.prototype.bind(..) 会创建一个新的包装函数，这个函数会忽略它当前的 this 绑定（无论绑定的对象是什么），并把我们提供的对象绑定到 this 上。

这样看起来硬绑定（也是显式绑定的一种）似乎比 new 绑定的优先级更高，无法使用 new 来控制 this 绑定。

我们看看是不是这样：

```
function foo(something) {
    this.a = something;
}

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar(3);
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

出乎意料！ bar 被硬绑定到 obj1 上，但是 new bar(3) 并没有像我们预计的那样把 obj1.a 修改为 3。相反，new 修改了硬绑定（到 obj1 的）调用 bar(..) 中的 this。因为使用了 new 绑定，我们得到了一个名字为 baz 的新对象，并且 baz.a 的值是 3。

再来看看我们之前介绍的“裸”辅助函数 bind：

```
function bind(fn, obj) {
    return function() {
        fn.apply( obj, arguments );
    };
}
```

非常令人惊讶，因为看起来在辅助函数中 new 操作符的调用无法修改 this 绑定，但是在刚才的代码中 new 确实修改了 this 绑定。

实际上，ES5 中内置的 Function.prototype.bind(..) 更加复杂。下面是 MDN 提供的一种 bind(..) 实现，为了方便阅读我们对代码进行了排版：

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== "function") {
            // 与 ECMAScript 5 最接近的
            // 内部 IsCallable 函数
            throw new TypeError(
                "Function.prototype.bind - what is trying " +
                "to be bound is not callable"
```

```
            );
        }

        var aArgs = Array.prototype.slice.call( arguments, 1 ),
            fToBind = this,
            fNOP = function(){},
            fBound = function(){
                return fToBind.apply(
                    (
                        this instanceof fNOP &&
                        oThis ? this : oThis
                    ),
                    aArgs.concat(
                       Array.prototype.slice.call( arguments )
                    );
            }
        ;

        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();

        return fBound;
    };
}
```

> 这种 bind(..) 是一种 polyfill 代码（polyfill 就是我们常说的刮墙用的腻子，polyfill 代码主要用于旧浏览器的兼容，比如说在旧的浏览器中并没有内置 bind 函数，因此可以使用 polyfill 代码在旧浏览器中实现新的功能），对于 new 使用的硬绑定函数来说，这段 polyfill 代码和 ES5 内置的 bind(..) 函数并不完全相同（后面会介绍为什么要在 new 中使用硬绑定函数）。由于 polyfill 并不是内置函数，所以无法创建一个不包含 .prototype 的函数，因此会具有一些副作用。如果你要在 new 中使用硬绑定函数并且依赖 polyfill 代码的话，一定要非常小心。

下面是 new 修改 this 的相关代码：

```
this instanceof fNOP &&
oThis ? this : oThis

// ... 以及：

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
```

我们并不会详细解释这段代码做了什么（这非常复杂并且不在我们的讨论范围之内），不过简单来说，这段代码会判断硬绑定函数是否是被 new 调用，如果是的话就会使用新创建的 this 替换硬绑定的 this。

那么，为什么要在 new 中使用硬绑定函数呢？直接使用普通函数不是更简单吗？

之所以要在 new 中使用硬绑定函数，主要目的是预先设置函数的一些参数，这样在使用 new 进行初始化时就可以只传入其余的参数。bind(..) 的功能之一就是可以把除了第一个参数（第一个参数用于绑定 this）之外的其他参数都传给下层的函数（这种技术称为"部分应用"，是"柯里化"的一种）。举例来说：

```
function foo(p1,p2) {
        this.val = p1 + p2;
}

// 之所以使用 null 是因为在本例中我们并不关心硬绑定的 this 是什么
// 反正使用 new 时 this 会被修改
var bar = foo.bind( null, "p1" );

var baz = new bar( "p2" );

baz.val; // p1p2
```

### 判断this

现在我们可以根据优先级来判断函数在某个调用位置应用的是哪条规则。可以按照下面的顺序来进行判断：

1. 函数是否在 new 中调用（new 绑定）？如果是的话 this 绑定的是新创建的对象。

   ```
   var bar = new foo()
   ```

2. 函数是否通过 call、apply（显式绑定）或者硬绑定调用？如果是的话，this 绑定的是指定的对象。

   ```
   var bar = foo.call(obj2)
   ```

3. 函数是否在某个上下文对象中调用（隐式绑定）？如果是的话，this 绑定的是那个上下文对象。

   ```
   var bar = obj1.foo()
   ```

4. 如果都不是的话，使用默认绑定。如果在严格模式下，就绑定到 undefined，否则绑定到全局对象。

   ```
   var bar = foo()
   ```

就是这样。对于正常的函数调用来说，理解了这些知识你就可以明白 this 的绑定原理了。不过……凡事总有例外。

## 2.4　绑定例外

规则总有例外，这里也一样。

在某些场景下 this 的绑定行为会出乎意料，你认为应当应用其他绑定规则时，实际上应用的可能是默认绑定规则。

## 2.4.1 被忽略的this

如果你把 null 或者 undefined 作为 this 的绑定对象传入 call、apply 或者 bind，这些值在调用时会被忽略，实际应用的是默认绑定规则：

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo.call( null ); // 2
```

那么什么情况下你会传入 null 呢？

一种非常常见的做法是使用 apply(..) 来“展开”一个数组，并当作参数传入一个函数。类似地，bind(..) 可以对参数进行柯里化（预先设置一些参数），这种方法有时非常有用：

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// 把数组“展开”成参数
foo.apply( null, [2, 3] ); // a:2, b:3

// 使用 bind(..) 进行柯里化
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

这两种方法都需要传入一个参数当作 this 的绑定对象。如果函数并不关心 this 的话，你仍然需要传入一个占位值，这时 null 可能是一个不错的选择，就像代码所示的那样。

> 尽管本书中未提到，但在 ES6 中，可以用 ... 操作符代替 apply(..) 来“展开”数组，foo(...[1,2]) 和 foo(1,2) 是一样的，这样可以避免不必要的 this 绑定。可惜，在 ES6 中没有柯里化的相关语法，因此还是需要使用 bind(..)。

然而，总是使用 null 来忽略 this 绑定可能产生一些副作用。如果某个函数确实使用了 this（比如第三方库中的一个函数），那默认绑定规则会把 this 绑定到全局对象（在浏览器中这个对象是 window），这将导致不可预计的后果（比如修改全局对象）。

显而易见，这种方式可能会导致许多难以分析和追踪的 bug。

**更安全的this**

一种“更安全”的做法是传入一个特殊的对象，把 this 绑定到这个对象不会对你的程序产生任何副作用。就像网络（以及军队）一样，我们可以创建一个“DMZ”（demilitarized zone，非军事区）对象——它就是一个空的非委托的对象（委托在第 5 章和第 6 章介绍）。

如果我们在忽略 this 绑定时总是传入一个 DMZ 对象，那就什么都不用担心了，因为任何对于 this 的使用都会被限制在这个空对象中，不会对全局对象产生任何影响。

由于这个对象完全是一个空对象，我自己喜欢用变量名 ø（这是数学中表示空集合符号的小写形式）来表示它。在大多数键盘（比如说 Mac 的 US 布局键盘）上都可以使用⌥ +o（Option-o）来打出这个符号。有些系统允许你为特殊符号设定快捷键。如果你不喜欢 ø 符号或者你的键盘不太容易打出这个符号，那你可以换一个喜欢的名字来称呼它。

无论你叫它什么，在 JavaScript 中创建一个空对象最简单的方法都是 Object.create(null)（详细介绍请看第 5 章）。Object.create(null) 和 {} 很像，但是并不会创建 Object.prototype 这个委托，所以它比 {}“更空”：

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// 我们的 DMZ 空对象
var ø = Object.create( null );

// 把数组展开成参数
foo.apply( ø, [2, 3] ); // a:2, b:3

// 使用 bind(..) 进行柯里化
var bar = foo.bind( ø, 2 );
bar( 3 ); // a:2, b:3
```

使用变量名 ø 不仅让函数变得更加“安全”，而且可以提高代码的可读性，因为 ø 表示“我希望 this 是空”，这比 null 的含义更清楚。不过再说一遍，你可以用任何喜欢的名字来命名 DMZ 对象。

## 2.4.2 间接引用

另一个需要注意的是，你有可能（有意或者无意地）创建一个函数的“间接引用”，在这种情况下，调用这个函数会应用默认绑定规则。

间接引用最容易在赋值时发生：

```
function foo() {
    console.log( this.a );
}
```

```
var a = 2;
var o = { a: 3, foo: foo };
var p = { a: 4 };

o.foo(); // 3
(p.foo = o.foo)(); // 2
```

赋值表达式 p.foo = o.foo 的返回值是目标函数的引用，因此调用位置是 foo() 而不是 p.foo() 或者 o.foo()。根据我们之前说过的，这里会应用默认绑定。

注意：对于默认绑定来说，决定 this 绑定对象的并不是调用位置是否处于严格模式，而是函数体是否处于严格模式。如果函数体处于严格模式，this 会被绑定到 undefined，否则 this 会被绑定到全局对象。

## 2.4.3 软绑定

之前我们已经看到过，硬绑定这种方式可以把 this 强制绑定到指定的对象（除了使用 new 时），防止函数调用应用默认绑定规则。问题在于，硬绑定会大大降低函数的灵活性，使用硬绑定之后就无法使用隐式绑定或者显式绑定来修改 this。

如果可以给默认绑定指定一个全局对象和 undefined 以外的值，那就可以实现和硬绑定相同的效果，同时保留隐式绑定或者显式绑定修改 this 的能力。

可以通过一种被称为软绑定的方法来实现我们想要的效果：

```
if (!Function.prototype.softBind) {
    Function.prototype.softBind = function(obj) {
        var fn = this;
        // 捕获所有 curried 参数
        var curried = [].slice.call( arguments, 1 );
        var bound = function() {
            return fn.apply(
                (!this || this === (window || global)) ?
                    obj : this
                curried.concat.apply( curried, arguments )
            );
        };
        bound.prototype = Object.create( fn.prototype );
        return bound;
    };
}
```

除了软绑定之外，softBind(..) 的其他原理和 ES5 内置的 bind(..) 类似。它会对指定的函数进行封装，首先检查调用时的 this，如果 this 绑定到全局对象或者 undefined，那就把指定的默认对象 obj 绑定到 this，否则不会修改 this。此外，这段代码还支持可选的柯里化（详情请查看之前和 bind(..) 相关的介绍）。

下面我们看看 softBind 是否实现了软绑定功能：

```
function foo() {
   console.log("name: " + this.name);
}

var obj = { name: "obj" },
    obj2 = { name: "obj2" },
    obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);
obj2.foo(); // name: obj2 <---- 看！！！

fooOBJ.call( obj3 ); // name: obj3 <---- 看!

setTimeout( obj2.foo, 10 );
// name: obj   <---- 应用了软绑定
```

可以看到，软绑定版本的 foo() 可以手动将 this 绑定到 obj2 或者 obj3 上，但如果应用默认绑定，则会将 this 绑定到 obj。

## 2.5 this词法

我们之前介绍的四条规则已经可以包含所有正常的函数。但是 ES6 中介绍了一种无法使用这些规则的特殊函数类型：箭头函数。

箭头函数并不是使用 function 关键字定义的，而是使用被称为“胖箭头”的操作符 => 定义的。箭头函数不使用 this 的四种标准规则，而是根据外层（函数或者全局）作用域来决定 this。

我们来看看箭头函数的词法作用域：

```
function foo() {
    // 返回一个箭头函数
    return (a) => {
        //this 继承自 foo()
        console.log( this.a );
    };
}

var obj1 = {
    a:2
};

var obj2 = {
    a:3
```

```
};

var bar = foo.call( obj1 );
bar.call( obj2 ); // 2，不是 3！
```

foo() 内部创建的箭头函数会捕获调用时 foo() 的 this。由于 foo() 的 this 绑定到 obj1，bar（引用箭头函数）的 this 也会绑定到 obj1，箭头函数的绑定无法被修改。（new 也不行！）

箭头函数最常用于回调函数中，例如事件处理器或者定时器：

```
function foo() {
    setTimeout(() => {
        // 这里的 this 在此法上继承自 foo()
        console.log( this.a );
    },100);
}

var obj = {
    a:2
};

foo.call( obj ); // 2
```

箭头函数可以像 bind(..) 一样确保函数的 this 被绑定到指定对象，此外，其重要性还体现在它用更常见的词法作用域取代了传统的 this 机制。实际上，在 ES6 之前我们就已经在使用一种几乎和箭头函数完全一样的模式。

```
function foo() {
    var self = this; // lexical capture of this
    setTimeout( function(){
        console.log( self.a );
    }, 100 );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

虽然 self = this 和箭头函数看起来都可以取代 bind(..)，但是从本质上来说，它们想替代的是 this 机制。

如果你经常编写 this 风格的代码，但是绝大部分时候都会使用 self = this 或者箭头函数来否定 this 机制，那你或许应当：

1. 只使用词法作用域并完全抛弃错误 this 风格的代码；
2. 完全采用 this 风格，在必要时使用 bind(..)，尽量避免使用 self = this 和箭头函数。

当然，包含这两种代码风格的程序可以正常运行，但是在同一个函数或者同一个程序中混合使用这两种风格通常会使代码更难维护，并且可能也会更难编写。

## 2.6 小结

如果要判断一个运行中函数的 `this` 绑定，就需要找到这个函数的直接调用位置。找到之后就可以顺序应用下面这四条规则来判断 `this` 的绑定对象。

1. 由 `new` 调用？绑定到新创建的对象。
2. 由 `call` 或者 `apply`（或者 `bind`）调用？绑定到指定的对象。
3. 由上下文对象调用？绑定到那个上下文对象。
4. 默认：在严格模式下绑定到 `undefined`，否则绑定到全局对象。

一定要注意，有些调用可能在无意中使用默认绑定规则。如果想“更安全”地忽略 `this` 绑定，你可以使用一个 DMZ 对象，比如 `ø = Object.create(null)`，以保护全局对象。

ES6 中的箭头函数并不会使用四条标准的绑定规则，而是根据当前的词法作用域来决定 `this`，具体来说，箭头函数会继承外层函数调用的 `this` 绑定（无论 `this` 绑定到什么）。这其实和 ES6 之前代码中的 `self = this` 机制一样。

第 3 章

# 对象

在第 1 章和第 2 章中，我们介绍了函数调用位置的不同会造成 `this` 绑定对象的不同。但是对象到底是什么，为什么我们需要绑定它们呢？本章会详细介绍对象。

## 3.1 语法

对象可以通过两种形式定义：声明（文字）形式和构造形式。

对象的文字语法大概是这样：

```
var myObj = {
    key: value
    // ...
};
```

构造形式大概是这样：

```
var myObj = new Object();
myObj.key = value;
```

构造形式和文字形式生成的对象是一样的。唯一的区别是，在文字声明中你可以添加多个键 / 值对，但是在构造形式中你必须逐个添加属性。

> 用上面的“构造形式”来创建对象是非常少见的，一般来说你会使用文字语法，绝大多数内置对象也是这样做的（稍后解释）。

## 3.2 类型

对象是 JavaScript 的基础。在 JavaScript 中一共有六种主要类型（术语是“语言类型”）：

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `object`

注意，简单基本类型（`string`、`boolean`、`number`、`null` 和 `undefined`）本身并不是对象。`null` 有时会被当作一种对象类型，但是这其实只是语言本身的一个 bug，即对 `null` 执行 `typeof null` 时会返回字符串 `"object"`。[1] 实际上，`null` 本身是基本类型。

有一种常见的错误说法是“JavaScript 中万物皆是对象”，这显然是错误的。

实际上，JavaScript 中有许多特殊的对象子类型，我们可以称之为复杂基本类型。

函数就是对象的一个子类型（从技术角度来说就是“可调用的对象”）。JavaScript 中的函数是“一等公民”，因为它们本质上和普通的对象一样（只是可以调用），所以可以像操作其他对象一样操作函数（比如当作另一个函数的参数）。

数组也是对象的一种类型，具备一些额外的行为。数组中内容的组织方式比一般的对象要稍微复杂一些。

### 内置对象

JavaScript 中还有一些对象子类型，通常被称为内置对象。有些内置对象的名字看起来和简单基础类型一样，不过实际上它们的关系更复杂，我们稍后会详细介绍。

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`

---

注 1：原理是这样的，不同的对象在底层都表示为二进制，在 JavaScript 中二进制前三位都为 0 的话会被判断为 `object` 类型，`null` 的二进制表示是全 0，自然前三位也是 0，所以执行 `typeof` 时会返回“`object`”。——译者注

- Date
- RegExp
- Error

这些内置对象从表现形式来说很像其他语言中的类型（type）或者类（class），比如 Java 中的 String 类。

但是在 JavaScript 中，它们实际上只是一些内置函数。这些内置函数可以当作构造函数（由 new 产生的函数调用——参见第 2 章）来使用，从而可以构造一个对应子类型的新对象。举例来说：

```
var strPrimitive = "I am a string";
typeof strPrimitive; // "string"
strPrimitive instanceof String; // false

var strObject = new String( "I am a string" );
typeof strObject; // "object"
strObject instanceof String; // true

// 检查 sub-type 对象
Object.prototype.toString.call( strObject ); // [object String]
```

在之后的章节中我们会详细介绍 Object.prototype.toString... 是如何工作的，不过简单来说，我们可以认为子类型在内部借用了 Object 中的 toString() 方法。从代码中可以看到，strObject 是由 String 构造函数创建的一个对象。

原始值 "I am a string" 并不是一个对象，它只是一个字面量，并且是一个不可变的值。如果要在这个字面量上执行一些操作，比如获取长度、访问其中某个字符等，那需要将其转换为 String 对象。

幸好，在必要时语言会自动把字符串字面量转换成一个 String 对象，也就是说你并不需要显式创建一个对象。JavaScript 社区中的大多数人都认为能使用文字形式时就不要使用构造形式。

思考下面的代码：

```
var strPrimitive = "I am a string";

console.log( strPrimitive.length ); // 13

console.log( strPrimitive.charAt( 3 ) ); // "m"
```

使用以上两种方法，我们都可以直接在字符串字面量上访问属性或者方法，之所以可以这样做，是因为引擎自动把字面量转换成 String 对象，所以可以访问属性和方法。

同样的事也会发生在数值字面量上，如果使用类似 42.359.toFixed(2) 的方法，引擎会把

42 转换成 new Number(42)。对于布尔字面量来说也是如此。

null 和 undefined 没有对应的构造形式，它们只有文字形式。相反，Date 只有构造，没有文字形式。

对于 Object、Array、Function 和 RegExp（正则表达式）来说，无论使用文字形式还是构造形式，它们都是对象，不是字面量。在某些情况下，相比用文字形式创建对象，构造形式可以提供一些额外选项。由于这两种形式都可以创建对象，所以我们首选更简单的文字形式。建议只在需要那些额外选项时使用构造形式。

Error 对象很少在代码中显式创建，一般是在抛出异常时被自动创建。也可以使用 new Error(..) 这种构造形式来创建，不过一般来说用不着。

## 3.3 内容

之前我们提到过，对象的内容是由一些存储在特定命名位置的（任意类型的）值组成的，我们称之为属性。

需要强调的一点是，当我们说“内容”时，似乎在暗示这些值实际上被存储在对象内部，但是这只是它的表现形式。在引擎内部，这些值的存储方式是多种多样的，一般并不会存在对象容器内部。存储在对象容器内部的是这些属性的名称，它们就像指针（从技术角度来说就是引用）一样，指向这些值真正的存储位置。

思考下面的代码：

```
var myObject = {
    a: 2
};

myObject.a; // 2

myObject["a"]; // 2
```

如果要访问 myObject 中 a 位置上的值，我们需要使用 . 操作符或者 [] 操作符。.a 语法通常被称为“属性访问”，["a"] 语法通常被称为“键访问”。实际上它们访问的是同一个位置，并且会返回相同的值 2，所以这两个术语是可以互换的。在本书中我们会使用最常见的术语“属性访问”。

这两种语法的主要区别在于 . 操作符要求属性名满足标识符的命名规范，而 [".."] 语法可以接受任意 UTF-8/Unicode 字符串作为属性名。举例来说，如果要引用名称为 "Super-Fun!" 的属性，那就必须使用 ["Super-Fun!"] 语法访问，因为 Super-Fun! 并不是一个有效的标识符属性名。

此外，由于 [".."] 语法使用字符串来访问属性，所以可以在程序中构造这个字符串，比如说：

```
var myObject = {
    a:2
};

var idx;

if (wantA) {
    idx = "a";
}

// 之后

console.log( myObject[idx] ); // 2
```

在对象中，属性名永远都是字符串。如果你使用 `string`（字面量）以外的其他值作为属性名，那它首先会被转换为一个字符串。即使是数字也不例外，虽然在数组下标中使用的的确是数字，但是在对象属性名中数字会被转换成字符串，所以当心不要搞混对象和数组中数字的用法：

```
var myObject = { };

myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";

myObject["true"]; // "foo"
myObject["3"]; // "bar"
myObject["[object Object]"]; // "baz"
```

### 3.3.1 可计算属性名

如果你需要通过表达式来计算属性名，那么我们刚刚讲到的 `myObject[..]` 这种属性访问语法就可以派上用场了，如可以使用 `myObject[prefix + name]`。但是使用文字形式来声明对象时这样做是不行的。

ES6 增加了*可计算属性名*，可以在文字形式中使用 `[]` 包裹一个表达式来当作属性名：

```
var prefix = "foo";

var myObject = {
    [prefix + "bar"]:"hello",
    [prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

可计算属性名最常用的场景可能是 ES6 的符号（Symbol），本书中不作详细介绍。不过简单来说，它们是一种新的基础数据类型，包含一个不透明且无法预测的值（从技术

角度来说就是一个字符串)。一般来说你不会用到符号的实际值(因为理论上来说在不同的 JavaScript 引擎中值是不同的),所以通常你接触到的是符号的名称,比如 Symbol.Something(这个名字是我编的):

```
var myObject = {
    [Symbol.Something]: "hello world"
}
```

## 3.3.2 属性与方法

如果访问的对象属性是一个函数,有些开发者喜欢使用不一样的叫法以作区分。由于函数很容易被认为是属于某个对象,在其他语言中,属于对象(也被称为"类")的函数通常被称为"方法",因此把"属性访问"说成是"方法访问"也就不奇怪了。

有意思的是,JavaScript 的语法规范也做出了同样的区分。

从技术角度来说,函数永远不会"属于"一个对象,所以把对象内部引用的函数称为"方法"似乎有点不妥。

确实,有些函数具有 this 引用,有时候这些 this 确实会指向调用位置的对象引用。但是这种用法从本质上来说并没有把一个函数变成一个"方法",因为 this 是在运行时根据调用位置动态绑定的,所以函数和对象的关系最多也只能说是间接关系。

无论返回值是什么类型,每次访问对象的属性就是属性访问。如果属性访问返回的是一个函数,那它也并不是一个"方法"。属性访问返回的函数和其他函数没有任何区别(除了可能发生的隐式绑定 this,就像我们刚才提到的)。

举例来说:

```
function foo() {
    console.log( "foo" );
}

var someFoo = foo; // 对 foo 的变量引用

var myObject = {
    someFoo: foo
};

foo; // function foo(){..}

someFoo; // function foo(){..}

myObject.someFoo; // function foo(){..}
```

someFoo 和 myObject.someFoo 只是对于同一个函数的不同引用,并不能说明这个函数是特别的或者"属于"某个对象。如果 foo() 定义时在内部有一个 this 引用,那这两个函数引

用的唯一区别就是 myObject.someFoo 中的 this 会被隐式绑定到一个对象。无论哪种引用形式都不能称之为“方法”。

或许有人会辩解说，函数并不是在定义时成为方法，而是在被调用时根据调用位置的不同（是否具有上下文对象——详见第 2 章）成为方法。即便如此，这种说法仍然有些不妥。

最保险的说法可能是，“函数”和“方法”在 JavaScript 中是可以互换的。

> ES6 增加了 super 引用，一般来说会被用在 class 中（参见附录 A）。super 的行为似乎更有理由把 super 绑定的函数称为“方法”。但是再说一次，这些只是一些语义（和技术）上的微妙差别，本质是一样的。

即使你在对象的文字形式中声明一个函数表达式，这个函数也不会“属于”这个对象——它们只是对于相同函数对象的多个引用。

```
var myObject = {
    foo: function() {
        console.log( "foo" );
    }
};

var someFoo = myObject.foo;

someFoo; // function foo(){..}

myObject.foo; // function foo(){..}
```

> 第 6 章会介绍本例对象的文字形式中声明函数的语法，这是 ES6 增加的一种简易函数声明语法。

### 3.3.3　数组

数组也支持 [] 访问形式，不过就像我们之前提到过的，数组有一套更加结构化的值存储机制（不过仍然不限制值的类型）。数组期望的是数值下标，也就是说值存储的位置（通常被称为索引）是整数，比如说 0 和 42：

```
var myArray = [ "foo", 42, "bar" ];

myArray.length; // 3

myArray[0]; // "foo"

myArray[2]; // "bar"
```

数组也是对象，所以虽然每个下标都是整数，你仍然可以给数组添加属性：

```
var myArray = [ "foo", 42, "bar" ];

myArray.baz = "baz";

myArray.length; // 3

myArray.baz; // "baz"
```

可以看到虽然添加了命名属性（无论是通过 . 语法还是 [] 语法），数组的 length 值并未发生变化。

你完全可以把数组当作一个普通的键 / 值对象来使用，并且不添加任何数值索引，但是这并不是一个好主意。数组和普通的对象都根据其对应的行为和用途进行了优化，所以最好只用对象来存储键 / 值对，只用数组来存储数值下标 / 值对。

注意：如果你试图向数组添加一个属性，但是属性名“看起来”像一个数字，那它会变成一个数值下标（因此会修改数组的内容而不是添加一个属性）：

```
var myArray = [ "foo", 42, "bar" ];

myArray["3"] = "baz";

myArray.length; // 4

myArray[3]; // "baz"
```

### 3.3.4　复制对象

JavaScript 初学者最常见的问题之一就是如何复制一个对象。看起来应该有一个内置的 copy() 方法，是吧？实际上事情比你想象的更复杂，因为我们无法选择一个默认的复制算法。

举例来说，思考一下这个对象：

```
function anotherFunction() { /*..*/ }

var anotherObject = {
    c: true
};

var anotherArray = [];

var myObject = {
    a: 2,
    b: anotherObject, // 引用，不是复本！
    c: anotherArray, // 另一个引用！
    d: anotherFunction
};
```

```
anotherArray.push( anotherObject, myObject );
```

如何准确地表示 myObject 的复制呢？

首先，我们应该判断它是浅复制还是深复制。对于浅拷贝来说，复制出的新对象中 a 的值会复制旧对象中 a 的值，也就是 2，但是新对象中 b、c、d 三个属性其实只是三个引用，它们和旧对象中 b、c、d 引用的对象是一样的。对于深复制来说，除了复制 myObject 以外还会复制 anotherObject 和 anotherArray。这时问题就来了，anotherArray 引用了 anotherObject 和 myObject，所以又需要复制 myObject，这样就会由于循环引用导致死循环。

我们是应该检测循环引用并终止循环（不复制深层元素）？还是应当直接报错或者是选择其他方法？

除此之外，我们还不确定“复制”一个函数意味着什么。有些人会通过 toString() 来序列化一个函数的源代码（但是结果取决于 JavaScript 的具体实现，而且不同的引擎对于不同类型的函数处理方式并不完全相同）。

那么如何解决这些棘手问题呢？许多 JavaScript 框架都提出了自己的解决办法，但是 JavaScript 应当采用哪种方法作为标准呢？在很长一段时间里，这个问题都没有明确的答案。

对于 JSON 安全（也就是说可以被序列化为一个 JSON 字符串并且可以根据这个字符串解析出一个结构和值完全一样的对象）的对象来说，有一种巧妙的复制方法：

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

当然，这种方法需要保证对象是 JSON 安全的，所以只适用于部分情况。

相比深复制，浅复制非常易懂并且问题要少得多，所以 ES6 定义了 Object.assign(..) 方法来实现浅复制。Object.assign(..) 方法的第一个参数是目标对象，之后还可以跟一个或多个源对象。它会遍历一个或多个源对象的所有可枚举（enumerable，参见下面的代码）的自有键（owned key，很快会介绍）并把它们复制（使用 = 操作符赋值）到目标对象，最后返回目标对象，就像这样：

```
var newObj = Object.assign( {}, myObject );

newObj.a; // 2
newObj.b === anotherObject; // true
newObj.c === anotherArray; // true
newObj.d === anotherFunction; // true
```

> 下一节会介绍“属性描述符”以及 Object.defineProperty(..) 的用法。但是需要注意的一点是，由于 Object.assign(..) 就是使用 = 操作符来赋值，所以源对象属性的一些特性（比如 writable）不会被复制到目标对象。

### 3.3.5 属性描述符

在 ES5 之前，JavaScript 语言本身并没有提供可以直接检测属性特性的方法，比如判断属性是否是只读。

但是从 ES5 开始，所有的属性都具备了属性描述符。

思考下面的代码：

```
var myObject = {
    a:2
};

Object.getOwnPropertyDescriptor( myObject, "a" );
// {
//    value: 2,
//    writable: true,
//    enumerable: true,
//    configurable: true
// }
```

如你所见，这个普通的对象属性对应的属性描述符（也被称为“数据描述符”，因为它只保存一个数据值）可不仅仅只是一个 2。它还包含另外三个特性：`writable`（可写）、`enumerable`（可枚举）和 `configurable`（可配置）。

在创建普通属性时属性描述符会使用默认值，我们也可以使用 `Object.defineProperty(..)` 来添加一个新属性或者修改一个已有属性（如果它是 `configurable`）并对特性进行设置。

举例来说：

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: true,
    enumerable: true
} );

myObject.a; // 2
```

我们使用 `defineProperty(..)` 给 `myObject` 添加了一个普通的属性并显式指定了一些特性。然而，一般来说你不会使用这种方式，除非你想修改属性描述符。

#### 1. Writable

`writable` 决定是否可以修改属性的值。

思考下面的代码：

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // 不可写!
    configurable: true,
    enumerable: true
} );

myObject.a = 3;

myObject.a; // 2
```

如你所见，我们对于属性值的修改静默失败（silently failed）了。如果在严格模式下，这种方法会出错：

```
"use strict";

var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // 不可写!
    configurable: true,
    enumerable: true
} );

myObject.a = 3; // TypeError
```

`TypeError` 错误表示我们无法修改一个不可写的属性。

> 之后我们会介绍 getter 和 setter，不过简单来说，你可以把 `writable:false` 看作是属性不可改变，相当于你定义了一个空操作 setter。严格来说，如果要和 `writable:false` 一致的话，你的 setter 被调用时应当抛出一个 `TypeError` 错误。

### 2. Configurable

只要属性是可配置的，就可以使用 `defineProperty(..)` 方法来修改属性描述符：

```
var myObject = {
    a:2
};

myObject.a = 3;
myObject.a; // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false, // 不可配置!
```

```
    enumerable: true
} );

myObject.a; // 4
myObject.a = 5;
myObject.a; // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

最后一个 defineProperty(..) 会产生一个 TypeError 错误，不管是不是处于严格模式，尝试修改一个不可配置的属性描述符都会出错。注意：如你所见，把 configurable 修改成 false 是单向操作，无法撤销！

> 要注意有一个小小的例外：即便属性是 configurable:false，我们还是可以把 writable 的状态由 true 改为 false，但是无法由 false 改为 true。

除了无法修改，configurable:false 还会禁止删除这个属性：

```
var myObject = {
    a:2
};

myObject.a; // 2

delete myObject.a;
myObject.a; // undefined

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: false,
    enumerable: true
} );

myObject.a; // 2
delete myObject.a;
myObject.a; // 2
```

如你所见，最后一个 delete 语句（静默）失败了，因为属性是不可配置的。

在本例中，delete 只用来直接删除对象的（可删除）属性。如果对象的某个属性是某个对象 / 函数的最后一个引用者，对这个属性执行 delete 操作之后，这个未引用的对象 / 函

数就可以被垃圾回收。但是，不要把 delete 看作一个释放内存的工具（就像 C/C++ 中那样），它就是一个删除对象属性的操作，仅此而已。

#### 3. Enumerable

这里我们要介绍的最后一个属性描述符（还有两个，我们会在介绍 getter 和 setter 时提到）是 enumerable。

从名字就可以看出，这个描述符控制的是属性是否会出现在对象的属性枚举中，比如说 for..in 循环。如果把 enumerable 设置成 false，这个属性就不会出现在枚举中，虽然仍然可以正常访问它。相对地，设置成 true 就会让它出现在枚举中。

用户定义的所有的普通属性默认都是 enumerable，这通常就是你想要的。但是如果你不希望某些特殊属性出现在枚举中，那就把它设置成 enumerable:false。

稍后我们会详细介绍可枚举性，这里先提示一下。

### 3.3.6　不变性

有时候你会希望属性或者对象是不可改变（无论有意还是无意）的，在 ES5 中可以通过很多种方法来实现。

很重要的一点是，所有的方法创建的都是浅不变形，也就是说，它们只会影响目标对象和它的直接属性。如果目标对象引用了其他对象（数组、对象、函数，等），其他对象的内容不受影响，仍然是可变的：

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

假设代码中的 myImmutableObject 已经被创建而且是不可变的，但是为了保护它的内容 myImmutableObject.foo，你还需要使用下面的方法让 foo 也不可变。

> 在 JavaScript 程序中很少需要深不可变性。有些特殊情况可能需要这样做，但是根据通用的设计模式，如果你发现需要密封或者冻结所有的对象，那你或许应当退一步，重新思考一下程序的设计，让它能更好地应对对象值的改变。

#### 1. 对象常量

结合 writable:false 和 configurable:false 就可以创建一个真正的常量属性（不可修改、重定义或者删除）：

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );
```

#### 2. 禁止扩展

如果你想禁止一个对象添加新属性并且保留已有属性，可以使用 `Object.preventExtensions(..)`：

```
var myObject = {
    a:2
};

Object.preventExtensions( myObject );

myObject.b = 3;
myObject.b; // undefined
```

在非严格模式下，创建属性 `b` 会静默失败。在严格模式下，将会抛出 `TypeError` 错误。

#### 3. 密封

`Object.seal(..)` 会创建一个“密封”的对象，这个方法实际上会在一个现有对象上调用 `Object.preventExtensions(..)` 并把所有现有属性标记为 `configurable:false`。

所以，密封之后不仅不能添加新属性，也不能重新配置或者删除任何现有属性（虽然可以修改属性的值）。

#### 4. 冻结

`Object.freeze(..)` 会创建一个冻结对象，这个方法实际上会在一个现有对象上调用 `Object.seal(..)` 并把所有“数据访问”属性标记为 `writable:false`，这样就无法修改它们的值。

这个方法是你可以应用在对象上的级别最高的不可变性，它会禁止对于对象本身及其任意直接属性的修改（不过就像我们之前说过的，这个对象引用的其他对象是不受影响的）。

你可以“深度冻结”一个对象，具体方法为，首先在这个对象上调用 `Object.freeze(..)`，然后遍历它引用的所有对象并在这些对象上调用 `Object.freeze(..)`。但是一定要小心，因为这样做有可能会在无意中冻结其他（共享）对象。

### 3.3.7 [[Get]]

属性访问在实现时有一个微妙却非常重要的细节，思考下面的代码：

```
var myObject = {
    a: 2
};

myObject.a; // 2
```

myObject.a 是一次属性访问，但是这条语句并不仅仅是在 myObjet 中查找名字为 a 的属性，虽然看起来好像是这样。

在语言规范中，myObject.a 在 myObject 上实际上是实现了 [[Get]] 操作（有点像函数调用：[[Get]]()）。对象默认的内置 [[Get]] 操作首先在对象中查找是否有名称相同的属性，如果找到就会返回这个属性的值。

然而，如果没有找到名称相同的属性，按照 [[Get]] 算法的定义会执行另外一种非常重要的行为。我们会在第 5 章中介绍这个行为（其实就是遍历可能存在的 [[Prototype]] 链，也就是原型链）。

如果无论如何都没有找到名称相同的属性，那 [[Get]] 操作会返回值 undefined：

```
var myObject = {
    a:2
};

myObject.b; // undefined
```

注意，这种方法和访问变量时是不一样的。如果你引用了一个当前词法作用域中不存在的变量，并不会像对象属性一样返回 undefined，而是会抛出一个 ReferenceError 异常：

```
var myObject = {
    a: undefined
};

myObject.a; // undefined

myObject.b; // undefined
```

从返回值的角度来说，这两个引用没有区别——它们都返回了 undefined。然而，尽管乍看之下没什么区别，实际上底层的 [[Get]] 操作对 myObject.b 进行了更复杂的处理。

由于仅根据返回值无法判断出到底变量的值为 undefined 还是变量不存在，所以 [[Get]] 操作返回了 undefined。不过稍后我们会介绍如何区分这两种情况。

### 3.3.8 [[Put]]

既然有可以获取属性值的 [[Get]] 操作，就一定有对应的 [[Put]] 操作。

你可能会认为给对象的属性赋值会触发 [[Put]] 来设置或者创建这个属性。但是实际情况

并不完全是这样。

[[Put]] 被触发时，实际的行为取决于许多因素，包括对象中是否已经存在这个属性（这是最重要的因素）。

如果已经存在这个属性，[[Put]] 算法大致会检查下面这些内容。

1. 属性是否是访问描述符（参见 3.3.9 节）？如果是并且存在 setter 就调用 setter。
2. 属性的数据描述符中 writable 是否是 false？如果是，在非严格模式下静默失败，在严格模式下抛出 TypeError 异常。
3. 如果都不是，将该值设置为属性的值。

如果对象中不存在这个属性，[[Put]] 操作会更加复杂。我们会在第 5 章讨论 [[Prototype]] 时详细进行介绍。

### 3.3.9 Getter和Setter

对象默认的 [[Put]] 和 [[Get]] 操作分别可以控制属性值的设置和获取。

> 在语言的未来 / 高级特性中，有可能可以改写整个对象（不仅仅是某个属性）的默认 [[Get]] 和 [[Put]] 操作。这已经超出了本书的讨论范围，但是将来“你不知道的 JavaScript”系列丛书中有可能会对这个问题进行探讨。

在 ES5 中可以使用 getter 和 setter 部分改写默认操作，但是只能应用在单个属性上，无法应用在整个对象上。getter 是一个隐藏函数，会在获取属性值时调用。setter 也是一个隐藏函数，会在设置属性值时调用。

当你给一个属性定义 getter、setter 或者两者都有时，这个属性会被定义为“访问描述符”（和“数据描述符”相对）。对于访问描述符来说，JavaScript 会忽略它们的 value 和 writable 特性，取而代之的是关心 set 和 get（还有 configurable 和 enumerable）特性。

思考下面的代码：

```
var myObject = {
    // 给 a 定义一个 getter
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject,  // 目标对象
    "b",         // 属性名
```

```
    {           // 描述符
        // 给 b 设置一个 getter
        get: function(){ return this.a * 2 },

        // 确保 b 会出现在对象的属性列表中
        enumerable: true
    }
);

myObject.a; // 2

myObject.b; // 4
```

不管是对象文字语法中的 `get a() { .. }`，还是 `defineProperty(..)` 中的显式定义，二者都会在对象中创建一个不包含值的属性，对于这个属性的访问会自动调用一个隐藏函数，它的返回值会被当作属性访问的返回值：

```
var myObject = {
    // 给 a 定义一个 getter
    get a() {
        return 2;
    }
};

myObject.a = 3;

myObject.a; // 2
```

由于我们只定义了 `a` 的 getter，所以对 `a` 的值进行设置时 `set` 操作会忽略赋值操作，不会抛出错误。而且即便有合法的 setter，由于我们自定义的 getter 只会返回 2，所以 `set` 操作是没有意义的。

为了让属性更合理，还应当定义 setter，和你期望的一样，setter 会覆盖单个属性默认的 `[[Put]]`（也被称为赋值）操作。通常来说 getter 和 setter 是成对出现的（只定义一个的话通常会产生意料之外的行为）：

```
var myObject = {
    // 给 a 定义一个 getter
    get a() {
        return this._a_;
    },

    // 给 a 定义一个 setter
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4
```

在本例中，实际上我们把赋值（[[Put]]）操作中的值 2 存储到了另一个变量 _a_ 中。名称 _a_ 只是一种惯例，没有任何特殊的行为——和其他普通属性一样。

## 3.3.10 存在性

前面我们介绍过，如 myObject.a 的属性访问返回值可能是 undefined，但是这个值有可能是属性中存储的 undefined，也可能是因为属性不存在所以返回 undefined。那么如何区分这两种情况呢？

我们可以在不访问属性值的情况下判断对象中是否存在这个属性：

```
var myObject = {
    a:2
};

("a" in myObject); // true
("b" in myObject); // false

myObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "b" ); // false
```

in 操作符会检查属性是否在对象及其 [[Prototype]] 原型链中（参见第 5 章）。相比之下，hasOwnProperty(..) 只会检查属性是否在 myObject 对象中，不会检查 [[Prototype]] 链。在第 5 章讲解 [[Prototype]] 时我们会详细介绍这两者的区别。

所有的普通对象都可以通过对于 Object.prototype 的委托（参见第 5 章）来访问 hasOwnProperty(..)，但是有的对象可能没有连接到 Object.prototype（通过 Object.create(null) 来创建——参见第 5 章）。在这种情况下，形如 myObejct.hasOwnProperty(..) 就会失败。

这时可以使用一种更加强硬的方法来进行判断：Object.prototype.hasOwnProperty.call(myObject,"a")，它借用基础的 hasOwnProperty(..) 方法并把它显式绑定（参见第 2 章）到 myObject 上。

看起来 in 操作符可以检查容器内是否有某个值，但是它实际上检查的是某个属性名是否存在。对于数组来说这个区别非常重要，4 in [2, 4, 6] 的结果并不是你期待的 True，因为 [2, 4, 6] 这个数组中包含的属性名是 0、1、2，没有 4。

### 1. 枚举

之前介绍 enumerable 属性描述符特性时我们简单解释过什么是“可枚举性”，现在详细介

绍一下：

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // 让 a 像普通属性一样可以枚举
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // 让 b 不可枚举
    { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true

// .......

for (var k in myObject) {
    console.log( k, myObject[k] );
}
// "a" 2
```

可以看到，myObject.b 确实存在并且有访问值，但是却不会出现在 for..in 循环中（尽管可以通过 in 操作符来判断是否存在）。原因是“可枚举”就相当于“可以出现在对象属性的遍历中”。

> 在数组上应用 for..in 循环有时会产生出人意料的结果，因为这种枚举不仅会包含所有数值索引，还会包含所有可枚举属性。最好只在对象上应用 for..in 循环，如果要遍历数组就使用传统的 for 循环来遍历数值索引。

也可以通过另一种方式来区分属性是否可枚举：

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // 让 a 像普通属性一样可以枚举
    { enumerable: true, value: 2 }
);

Object.defineProperty(
```

```
        myObject,
        "b",
        // 让 b 不可枚举
        { enumerable: false, value: 3 }
    );

    myObject.propertyIsEnumerable( "a" ); // true
    myObject.propertyIsEnumerable( "b" ); // false

    Object.keys( myObject ); // ["a"]
    Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

propertyIsEnumerable(..) 会检查给定的属性名是否直接存在于对象中（而不是在原型链上）并且满足 enumerable:true。

Object.keys(..) 会返回一个数组，包含所有可枚举属性，Object.getOwnPropertyNames(..) 会返回一个数组，包含所有属性，无论它们是否可枚举。

in 和 hasOwnProperty(..) 的区别在于是否查找 [[Prototype]] 链，然而，Object.keys(..) 和 Object.getOwnPropertyNames(..) 都只会查找对象直接包含的属性。

（目前）并没有内置的方法可以获取 in 操作符使用的属性列表（对象本身的属性以及 [[Prototype]] 链中的所有属性，参见第 5 章）。不过你可以递归遍历某个对象的整条 [[Prototype]] 链并保存每一层中使用 Object.keys(..) 得到的属性列表——只包含可枚举属性。

## 3.4 遍历

for..in 循环可以用来遍历对象的可枚举属性列表（包括 [[Prototype]] 链）。但是如何遍历属性的值呢？

对于数值索引的数组来说，可以使用标准的 for 循环来遍历值：

```
var myArray = [1, 2, 3];

for (var i = 0; i < myArray.length; i++) {
    console.log( myArray[i] );
}
// 1 2 3
```

这实际上并不是在遍历值，而是遍历下标来指向值，如 myArray[i]。

ES5 中增加了一些数组的辅助迭代器，包括 forEach(..)、every(..) 和 some(..)。每种辅助迭代器都可以接受一个回调函数并把它应用到数组的每个元素上，唯一的区别就是它们对于回调函数返回值的处理方式不同。

forEach(..) 会遍历数组中的所有值并忽略回调函数的返回值。every(..) 会一直运行直到

回调函数返回 false（或者“假”值），some(..) 会一直运行直到回调函数返回 true（或者“真”值）。

every(..) 和 some(..) 中特殊的返回值和普通 for 循环中的 break 语句类似，它们会提前终止遍历。

使用 for..in 遍历对象是无法直接获取属性值的，因为它实际上遍历的是对象中的所有可枚举属性，你需要手动获取属性值。

> 遍历数组下标时采用的是数字顺序（for 循环或者其他迭代器），但是遍历对象属性时的顺序是不确定的，在不同的 JavaScript 引擎中可能不一样。因此，在不同的环境中需要保证一致性时，一定不要相信任何观察到的顺序，它们是不可靠的。

那么如何直接遍历值而不是数组下标（或者对象属性）呢？幸好，ES6 增加了一种用来遍历数组的 for..of 循环语法（如果对象本身定义了迭代器的话也可以遍历对象）：

```
var myArray = [ 1, 2, 3 ];

for (var v of myArray) {
    console.log( v );
}
// 1
// 2
// 3
```

for..of 循环首先会向被访问对象请求一个迭代器对象，然后通过调用迭代器对象的 next() 方法来遍历所有返回值。

数组有内置的 @@iterator，因此 for..of 可以直接应用在数组上。我们使用内置的 @@iterator 来手动遍历数组，看看它是怎么工作的：

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();

it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { done:true }
```

> 我们使用 ES6 中的符号 Symbol.iterator 来获取对象的 @@iterator 内部属性。之前我们简单介绍过符号（Symbol，参见 3.3.1 节），跟这里的原理是相同的。引用类似 iterator 的特殊属性时要使用符号名，而不是符号包含的值。此外，虽然看起来很像一个对象，但是 @@iterator 本身并不是一个迭代器对象，而是一个返回迭代器对象的函数——这点非常精妙并且非常重要。

如你所见，调用迭代器的 `next()` 方法会返回形式为 `{ value: .. , done: .. }` 的值，`value` 是当前的遍历值，`done` 是一个布尔值，表示是否还有可以遍历的值。

注意，和值“3”一起返回的是 `done:false`，乍一看好像很奇怪，你必须再调用一次 `next()` 才能得到 `done:true`，从而确定完成遍历。这个机制和 ES6 中发生器函数的语义相关，不过已经超出了我们的讨论范围。

和数组不同，普通的对象没有内置的 `@@iterator`，所以无法自动完成 `for..of` 遍历。之所以要这样做，有许多非常复杂的原因，不过简单来说，这样做是为了避免影响未来的对象类型。

当然，你可以给任何想遍历的对象定义 `@@iterator`，举例来说：

```
var myObject = {
    a: 2,
    b: 3
};

Object.defineProperty( myObject, Symbol.iterator, {
    enumerable: false,
    writable: false,
    configurable: true,
    value: function() {
        var o = this;
        var idx = 0;
        var ks = Object.keys( o );
        return {
            next: function() {
                return {
                    value: o[ks[idx++]],
                    done: (idx > ks.length)
                };
            }
        };
    }
} );

// 手动遍历 myObject
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }

// 用 for..of 遍历 myObject
for (var v of myObject) {
    console.log( v );
}
// 2
// 3
```

我们使用 `Object.defineProperty(..)` 定义了我们自己的 `@@iterator`（主要是为了让它不可枚举），不过注意，我们把符号当作可计算属性名（本章之前有介绍）。此外，也可以直接在定义对象时进行声明，比如 `var myObject = { a:2, b:3, [Symbol.iterator]: function() { /* .. */ } }`。

`for..of` 循环每次调用 `myObject` 迭代器对象的 `next()` 方法时，内部的指针都会向前移动并返回对象属性列表的下一个值（再次提醒，需要注意遍历对象属性 / 值时的顺序）。

代码中的遍历非常简单，只是传递了属性本身的值。不过只要你愿意，当然也可以在自定义的数据结构上实现各种复杂的遍历。对于用户定义的对象来说，结合 `for..of` 循环和自定义迭代器可以组成非常强大的对象操作工具。

比如说，一个 `Pixel` 对象（有 x 和 y 坐标值）列表可以按照距离原点的直线距离来决定遍历顺序，也可以过滤掉“太远”的点，等等。只要迭代器的 `next()` 调用会返回 `{ value: .. }` 和 `{ done: true }`，ES6 中的 `for..of` 就可以遍历它。

实际上，你甚至可以定义一个“无限”迭代器，它永远不会“结束”并且总会返回一个新值（比如随机数、递增值、唯一标识符，等等）。你可能永远不会在 `for..of` 循环中使用这样的迭代器，因为它永远不会结束，你的程序会被挂起：

```
var randoms = {
    [Symbol.iterator]: function() {
        return {
            next: function() {
                return { value: Math.random() };
            }
        };
    }
};

var randoms_pool = [];
for (var n of randoms) {
    randoms_pool.push( n );

    // 防止无限运行!
    if (randoms_pool.length === 100) break;
}
```

这个迭代器会生成“无限个”随机数，因此我们添加了一条 `break` 语句，防止程序被挂起。

# 3.5 小结

JavaScript 中的对象有字面形式（比如 `var a = { .. }`）和构造形式（比如 `var a = new Array(..)`）。字面形式更常用，不过有时候构造形式可以提供更多选项。

许多人都以为“JavaScript 中万物都是对象”，这是错误的。对象是 6 个（或者是 7 个，取决于你的观点）基础类型之一。对象有包括 function 在内的子类型，不同子类型具有不同的行为，比如内部标签 [object Array] 表示这是对象的子类型数组。

对象就是键 / 值对的集合。可以通过 .propName 或者 ["propName"] 语法来获取属性值。访问属性时，引擎实际上会调用内部的默认 [[Get]] 操作（在设置属性值时是 [[Put]]），[[Get]] 操作会检查对象本身是否包含这个属性，如果没找到的话还会查找 [[Prototype]] 链（参见第 5 章）。

属性的特性可以通过属性描述符来控制，比如 writable 和 configurable。此外，可以使用 Object.preventExtensions(..)、Object.seal(..) 和 Object.freeze(..) 来设置对象（及其属性）的不可变性级别。

属性不一定包含值——它们可能是具备 getter/setter 的“访问描述符”。此外，属性可以是可枚举或者不可枚举的，这决定了它们是否会出现在 for..in 循环中。

你可以使用 ES6 的 for..of 语法来遍历数据结构（数组、对象，等等）中的值，for..of 会寻找内置或者自定义的 @@iterator 对象并调用它的 next() 方法来遍历数据值。

# 第 4 章

# 混合对象“类”

上一章介绍了对象，这章自然要介绍和类相关的面向对象编程。在研究类的具体机制之前，我们首先会介绍面向类的设计模式：实例化（instantiation）、继承（inheritance）和（相对）多态（polymorphism）。

我们将会看到，这些概念实际上无法直接对应到 JavaScript 的对象机制，因此我们会介绍许多 JavaScript 开发者所使用的解决方法（比如混入，mixin）。

> 本章用很大的篇幅（整整半章）介绍面向对象编程理论。在后半章介绍混入时会把这些概念落实到 JavaScript 代码上。但是首先我们会看到许多概念和伪代码，因此千万不要迷路——跟紧了！

## 4.1 类理论

类 / 继承描述了一种代码的组织结构形式——一种在软件中对真实世界中问题领域的建模方法。

面向对象编程强调的是数据和操作数据的行为本质上是互相关联的（当然，不同的数据有不同的行为），因此好的设计就是把数据以及和它相关的行为打包（或者说封装）起来。这在正式的计算机科学中有时被称为数据结构。

举例来说，用来表示一个单词或者短语的一串字符通常被称为字符串。字符就是数据。但是你关心的往往不是数据是什么，而是可以对数据做什么，所以可以应用在这种数据上的行为（计算长度、添加数据、搜索，等等）都被设计成 `String` 类的方法。

所有字符串都是 `String` 类的一个实例，也就是说它是一个包裹，包含字符数据和我们可以应用在数据上的函数。

我们还可以使用类对数据结构进行分类，可以把任意数据结构看作范围更广的定义的一种特例。

我们来看一个常见的例子，“汽车”可以被看作“交通工具”的一种特例，后者是更广泛的类。

我们可以在软件中定义一个 `Vehicle` 类和一个 `Car` 类来对这种关系进行建模。

`Vehicle` 的定义可能包含推进器（比如引擎）、载人能力等等，这些都是 `Vehicle` 的行为。我们在 `Vehicle` 中定义的是（几乎）所有类型的交通工具（飞机、火车和汽车）都包含的东西。

在我们的软件中，对不同的交通工具重复定义“载人能力”是没有意义的。相反，我们只在 `Vehicle` 中定义一次，定义 `Car` 时，只要声明它继承（或者扩展）了 `Vehicle` 的这个基础定义就行。`Car` 的定义就是对通用 `Vehicle` 定义的特殊化。

虽然 `Vehicle` 和 `Car` 会定义相同的方法，但是实例中的数据可能是不同的，比如每辆车独一无二的 VIN（Vehicle Identification Number，车辆识别号码），等等。

这就是类、继承和实例化。

类的另一个核心概念是*多态*，这个概念是说父类的通用行为可以被子类用更特殊的行为重写。实际上，相对多态性允许我们从重写行为中引用基础行为。

类理论强烈建议父类和子类使用相同的方法名来表示特定的行为，从而让子类重写父类。我们之后会看到，在 JavaScript 代码中这样做会降低代码的可读性和健壮性。

### 4.1.1 “类”设计模式

你可能从来没把类作为设计模式来看待，讨论得最多的是面向对象设计模式，比如*迭代器模式*、*观察者模式*、*工厂模式*、*单例模式*，等等。从这个角度来说，我们似乎是在（低级）面向对象类的基础上实现了*所有*（高级）设计模式，似乎面向对象是优秀代码的基础。

如果你之前接受过正规的编程教育的话，可能听说过*过程化编程*，这种代码只包含过程（函数）调用，没有高层的抽象。或许老师还教过你最好使用类把过程化风格的“意大利面代码”转换成结构清晰、组织良好的代码。

当然，如果你有*函数式编程*（比如 Monad）的经验就会知道类也是非常常用的一种设计模式。但是对于其他人来说，这可能是第一次知道类并不是必须的编程基础，而是一种可选的代码抽象。

有些语言（比如 Java）并不会给你选择的机会，类并不是可选的——万物皆是类。其他语言（比如 C/C++ 或者 PHP）会提供过程化和面向类这两种语法，开发者可以选择其中一种风格或者混用两种风格。

### 4.1.2 JavaScript中的“类”

JavaScript 属于哪一类呢？在相当长的一段时间里，JavaScript 只有一些近似类的语法元素（比如 `new` 和 `instanceof`），不过在后来的 ES6 中新增了一些元素，比如 `class` 关键字（参见附录 A）。

这是不是意味着 JavaScript 中实际上有类呢？简单来说：不是。

由于类是一种设计模式，所以你可以用一些方法（本章之后会介绍）近似实现类的功能。为了满足对于类设计模式的最普遍需求，JavaScript 提供了一些近似类的语法。

虽然有近似类的语法，但是 JavaScript 的机制似乎一直在阻止你使用类设计模式。在近似类的表象之下，JavaScript 的机制其实和类完全不同。语法糖和（广泛使用的）JavaScript“类”库试图掩盖这个现实，但是你迟早会面对它：其他语言中的类和 JavaScript 中的“类”并不一样。

总结一下，在软件设计中类是一种可选的模式，你需要自己决定是否在 JavaScript 中使用它。由于许多开发者都非常喜欢面向类的软件设计，我们会在本章的剩余部分中介绍如何在 JavaScript 中实现类以及存在的一些问题。

## 4.2 类的机制

在许多面向类的语言中，“标准库”会提供 `Stack` 类，它是一种“栈”数据结构（支持压入、弹出，等等）。`Stack` 类内部会有一些变量来存储数据，同时会提供一些公有的可访问行为（“方法”），从而让你的代码可以和（隐藏的）数据进行交互（比如添加、删除数据）。

但是在这些语言中，你实际上并不是直接操作 `Stack`（除非创建一个静态类成员引用，这超出了我们的讨论范围）。`Stack` 类仅仅是一个抽象的表示，它描述了所有“栈”需要做的事，但是它本身并不是一个“栈”。你必须先实例化 `Stack` 类然后才能对它进行操作。

### 4.2.1 建造

“类”和“实例”的概念来源于房屋建造。

建筑师会规划出一个建筑的所有特性：多宽、多高、多少个窗户以及窗户的位置，甚至连建造墙和房顶需要的材料都要计划好。在这个阶段他并不需要关心建筑会被建在哪，也不需要关心会建造多少个这样的建筑。

建筑师也不太关心建筑里的内容——家具、壁纸、吊扇等——他只关心需要用什么结构来容纳它们。

建筑蓝图只是建筑计划，它们并不是真正的建筑，我们还需要一个建筑工人来建造建筑。建筑工人会按照蓝图建造建筑。实际上，他会把规划好的特性从蓝图中复制到现实世界的建筑中。

完成后，建筑就成为了蓝图的物理实例，本质上就是对蓝图的复制。之后建筑工人就可以到下一个地方，把所有工作都重复一遍，再创建一份副本。

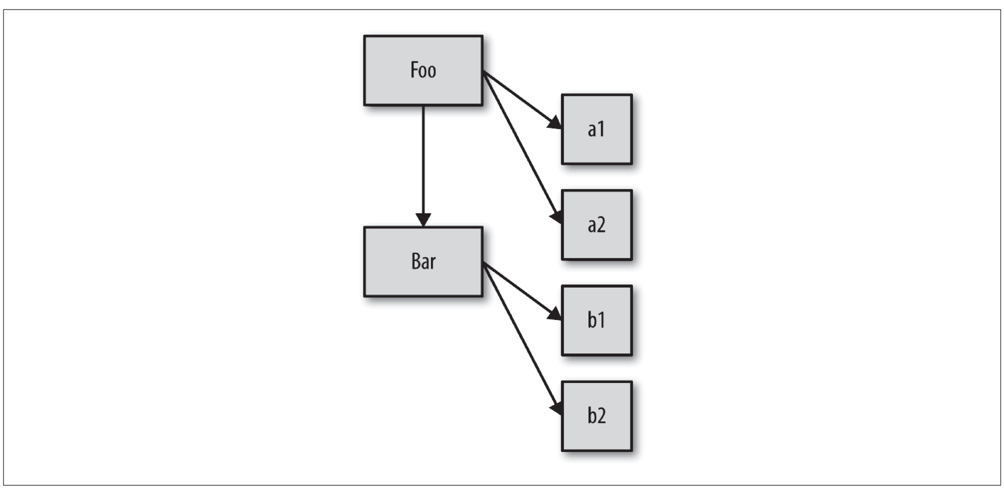
建筑和蓝图之间的关系是间接的。你可以通过蓝图了解建筑的结构，只观察建筑本身是无法获得这些信息的。但是如果你想打开一扇门，那就必须接触真实的建筑才行——蓝图只能表示门应该在哪，但并不是真正的门。

一个类就是一张蓝图。为了获得真正可以交互的对象，我们必须按照类来建造（也可以说实例化）一个东西，这个东西通常被称为实例，有需要的话，我们可以直接在实例上调用方法并访问其所有公有数据属性。

这个对象就是类中描述的所有特性的一份副本。

你走进一栋建筑时，它的蓝图不太可能挂在墙上（尽管这个蓝图可能会保存在公共档案馆中）。类似地，你通常也不会使用一个实例对象来直接访问并操作它的类，不过至少可以判断出这个实例对象来自哪个类。

把类和实例对象之间的关系看作是直接关系而不是间接关系通常更有助于理解。类通过复制操作被实例化为对象形式：




如你所见，箭头的方向是从左向右、从上向下，它表示概念和物理意义上发生的复制操作。

### 4.2.2 构造函数

类实例是由一个特殊的类方法构造的，这个方法名通常和类名相同，被称为构造函数。这个方法的任务就是初始化实例需要的所有信息（状态）。

举例来说，思考下面这个关于类的伪代码（编造出来的语法）：

```
class CoolGuy {
    specialTrick = nothing

    CoolGuy( trick ) {
        specialTrick = trick
    }

    showOff() {
        output( "Here's my trick: ", specialTrick )
    }
}
```

我们可以调用类构造函数来生成一个 CoolGuy 实例：

```
Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // 这是我的绝技：跳绳
```

注意，CoolGuy 类有一个 CoolGuy() 构造函数，执行 new CoolGuy() 时实际上调用的就是它。构造函数会返回一个对象（也就是类的一个实例），之后我们可以在这个对象上调用 showOff() 方法，来输出指定 CoolGuy 的特长。

显然，跳绳让乔成为了一个非常酷的家伙。

类构造函数属于类，而且通常和类同名。此外，构造函数大多需要用 new 来调，这样语言引擎才知道你想要构造一个新的类实例。

## 4.3 类的继承

在面向类的语言中，你可以先定义一个类，然后定义一个继承前者的类。

后者通常被称为“子类”，前者通常被称为“父类”。这些术语显然是类比父母和孩子，不过在意思上稍有扩展，你很快就会看到。

对于父母的亲生孩子来说，父母的基因特性会被复制给孩子。显然，在大多数生物的繁殖系统中，双亲都会贡献等量的基因给孩子。但是在编程语言中，我们假设只有一个父类。

一旦孩子出生，他们就变成了单独的个体。虽然孩子会从父母继承许多特性，但是他是一个独一无二的存在。如果孩子的头发是红色，父母的头发未必是红的，也不会随之变红，

二者之间没有直接的联系。

同理，定义好一个子类之后，相对于父类来说它就是一个独立并且完全不同的类。子类会包含父类行为的原始副本，但是也可以重写所有继承的行为甚至定义新行为。

非常重要的一点是，我们讨论的父类和子类并不是实例。父类和子类的比喻容易造成一些误解，实际上我们应当把父类和子类称为父类 DNA 和子类 DNA。我们需要根据这些 DNA 来创建（或者说实例化）一个人，然后才能和他进行沟通。

好了，我们先抛开现实中的父母和孩子，来看一个稍有不同的例子：不同类型的交通工具。这是一个非常典型（并且经常被抱怨）的讲解继承的例子。

首先回顾一下本章前面部分提出的 Vehicle 和 Car 类。思考下面关于类继承的伪代码：

```
class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." );
    }

    drive() {
        ignition();
        output( "Steering and moving forward!" )
    }
}

class Car inherits Vehicle {
    wheels = 4

    drive() {
        inherited:drive()
        output( "Rolling on all ", wheels, " wheels!" )
    }
}

class SpeedBoat inherits Vehicle {
    engines = 2

    ignition() {
        output( "Turning on my ", engines, " engines." )
    }

    pilot() {
        inherited:drive()
        output( "Speeding through the water with ease!" )
    }
}
```

为了方便理解并缩短代码，我们省略了这些类的构造函数。

我们通过定义 `Vehicle` 类来假设一种发动机，一种点火方式，一种驾驶方法。但是你不可能制造一个通用的“交通工具”，因为这个类只是一个抽象的概念。

接下来我们定义了两类具体的交通工具：`Car` 和 `SpeedBoat`。它们都从 `Vehicle` 继承了通用的特性并根据自身类别修改了某些特性。汽车需要四个轮子，快艇需要两个发动机，因此它必须启动两个发动机的点火装置。

## 4.3.1 多态

`Car` 重写了继承自父类的 `drive()` 方法，但是之后 `Car` 调用了 `inherited:drive()` 方法，这表明 `Car` 可以引用继承来的原始 `drive()` 方法。快艇的 `pilot()` 方法同样引用了原始 `drive()` 方法。

这个技术被称为*多态*或者*虚拟多态*。在本例中，更恰当的说法是*相对多态*。

多态是一个非常广泛的话题，我们现在所说的“相对”只是多态的一个方面：任何方法都可以引用继承层次中高层的方法（无论高层的方法名和当前方法名是否相同）。之所以说“相对”是因为我们并不会定义想要访问的绝对继承层次（或者说类），而是使用相对引用“查找上一层”。

在许多语言中可以使用 `super` 来代替本例中的 `inherited:`，它的含义是“超类”（superclass），表示当前类的父类 / 祖先类。

多态的另一个方面是，在继承链的不同层次中一个方法名可以被多次定义，当调用方法时会自动选择合适的定义。

在之前的代码中就有两个这样的例子：`drive()` 被定义在 `Vehicle` 和 `Car` 中，`ignition()` 被定义在 `Vehicle` 和 `SpeedBoat` 中。

在传统的面向类的语言中 `super` 还有一个功能，就是从子类的构造函数中通过 `super` 可以直接调用父类的构造函数。通常来说这没什么问题，因为对于真正的类来说，构造函数是属于类的。然而，在 JavaScript 中恰好相反——实际上“类”是属于构造函数的（类似 `Foo.prototype...` 这样的类型引用）。由于 JavaScript 中父类和子类的关系只存在于两者构造函数对应的 `.prototype` 对象中，因此它们的构造函数之间并不存在直接联系，从而无法简单地实现两者的相对引用（在 ES6 的类中可以通过 `super` 来“解决”这个问题，参见附录 A）。

我们可以在 ignition() 中看到多态非常有趣的一点。在 pilot() 中通过相对多态引用了（继承来的）Vehicle 中的 drive()。但是那个 drive() 方法直接通过名字（而不是相对引用）引用了 ignotion() 方法。

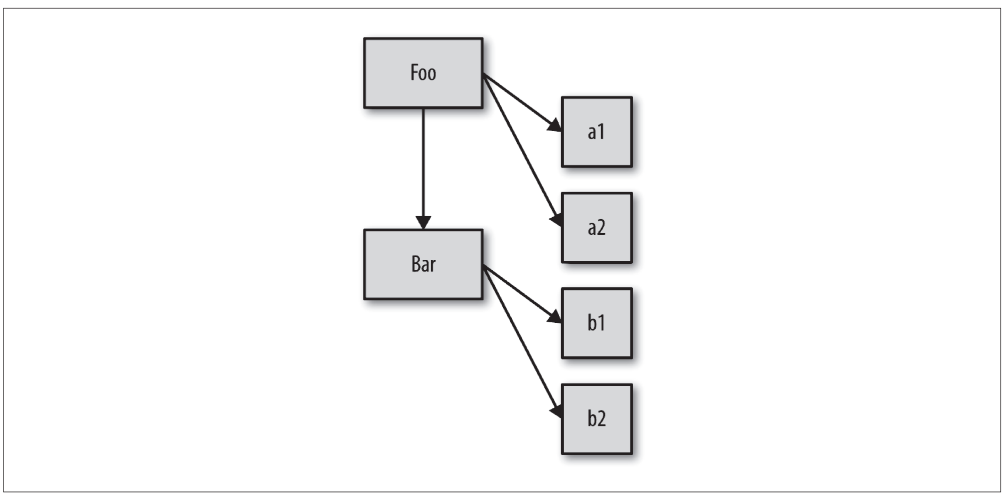
那么语言引擎会使用哪个 ignition() 呢，Vehicle 的还是 SpeedBoat 的？实际上它会使用 SpeedBoat 的 ignition()。如果你直接实例化了 Vehicle 类然后调用它的 drive()，那语言引擎就会使用 Vehicle 中的 ignition() 方法。

换言之，ignition() 方法定义的多态性取决于你是在哪个类的实例中引用它。

这似乎是一个过于深入的学术细节，但是只有理解了这个细节才能理解 JavaScript 中类似（但是并不相同）的 [[Prototype]] 机制。

在子类（而不是它们创建的实例对象！）中也可以相对引用它继承的父类，这种相对引用通常被称为 super。

还记得之前的那张图吗？




注意这些实例（a1、a2、b1 和 b2）和继承（Bar），箭头表示复制操作。

从概念上来说，子类 Bar 应当可以通过相对多态引用（或者说 super）来访问父类 Foo 中的行为。需要注意，子类得到的仅仅是继承自父类行为的一份副本。子类对继承到的一个方法进行“重写”，不会影响父类中的方法，这两个方法互不影响，因此才能使用相对多态引用访问父类中的方法（如果重写会影响父类的方法，那重写之后父类中的原始方法就不存在了，自然也无法引用）。

多态并不表示子类和父类有关联，子类得到的只是父类的一份副本。类的继承其实就是复制。

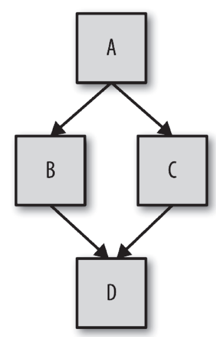
### 4.3.2 多重继承

还记得我们之前关于父类、子类和 DNA 的讨论吗？当时我们说这个比喻不太恰当，因为在现实中绝大多数后代是由双亲产生的。如果类可以继承两个类，那看起来就更符合现实的比喻了。

有些面向类的语言允许你继承多个“父类”。多重继承意味着所有父类的定义都会被复制到子类中。

从表面上来，对于类来说这似乎是一个非常有用的功能，可以把许多功能组合在一起。然而，这个机制同时也会带来很多复杂的问题。如果两个父类中都定义了 `drive()` 方法的话，子类引用的是哪个呢？难道每次都需要手动指定具体父类的 `drive()` 方法吗？这样多态继承的很多优点就存在了。

除此之外，还有一种被称为*钻石问题*的变种。在钻石问题中，子类 `D` 继承自两个父类（`B` 和 `C`），这两个父类都继承自 `A`。如果 `A` 中有 `drive()` 方法并且 `B` 和 `C` 都重写了这个方法（多态），那当 `D` 引用 `drive()` 时应当选择哪个版本呢（`B:drive()` 还是 `C:drive()`）？




这些问题远比看上去要复杂得多。之所以要介绍这些问题，主要是为了和 JavaScript 的机制进行对比。

相比之下，JavaScript 要简单得多：它本身并不提供“多重继承”功能。许多人认为这是件好事，因为使用多重继承的代价太高。然而这无法阻挡开发者们的热情，他们会尝试各种各样的办法来实现多重继承，我们马上就会看到。

## 4.4 混入

在继承或者实例化时，JavaScript 的对象机制并不会*自动*执行复制行为。简单来说，JavaScript 中只有对象，并不存在可以被实例化的“类”。一个对象并不会被复制到其他对象，它们会被*关联起来*（参见第 5 章）。

由于在其他语言中类表现出来的都是复制行为，因此 JavaScript 开发者也想出了一个方法来模拟类的复制行为，这个方法就是混入。接下来我们会看到两种类型的混入：显式和隐式。

### 4.4.1　显式混入

首先我们来回顾一下之前提到的 `Vehicle` 和 `Car`。由于 JavaScript 不会自动实现 `Vehicle` 到 `Car` 的复制行为，所以我们需要手动实现复制功能。这个功能在许多库和框架中被称为 `extend(..)`，但是为了方便理解我们称之为 `mixin(..)`。

```
// 非常简单的 mixin(..) 例子 :
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // 只会在不存在的情况下复制
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log(
           "Rolling on all " + this.wheels + " wheels!"
        );
    }
} );
```

有一点需要注意，我们处理的已经不再是类了，因为在 JavaScript 中不存在类，`Vehicle` 和 `Car` 都是对象，供我们分别进行复制和粘贴。

现在 Car 中就有了一份 Vehicle 属性和函数的副本了。从技术角度来说，函数实际上没有被复制，复制的是函数引用。所以，Car 中的属性 ignition 只是从 Vehicle 中复制过来的对于 ignition() 函数的引用。相反，属性 engines 就是直接从 Vehicle 中复制了值 1。

Car 已经有了 drive 属性（函数），所以这个属性引用并没有被 mixin 重写，从而保留了 Car 中定义的同名属性，实现了“子类”对“父类”属性的重写（参见 mixin(..) 例子中的 if 语句）。

### 1. 再说多态

我们来分析一下这条语句：Vehicle.drive.call( this )。这就是我所说的显式多态。还记得吗，在之前的伪代码中对应的语句是 inherited:drive()，我们称之为相对多态。

JavaScript（在 ES6 之前；参见附录 A）并没有相对多态的机制。所以，由于 Car 和 Vehicle 中都有 drive() 函数，为了指明调用对象，我们必须使用绝对（而不是相对）引用。我们通过名称显式指定 Vehicle 对象并调用它的 drive() 函数。

但是如果直接执行 Vehicle.drive()，函数调用中的 this 会被绑定到 Vehicle 对象而不是 Car 对象（参见第 2 章），这并不是我们想要的。因此，我们会使用 .call(this)（参见第 2 章）来确保 drive() 在 Car 对象的上下文中执行。

> 如果函数 Car.drive() 的名称标识符并没有和 Vehicle.drive() 重叠（或者说“屏蔽”；参见第 5 章）的话，我们就不需要实现方法多态，因为调用 mixin(..) 时会把函数 Vehicle.drive() 的引用复制到 Car 中，因此我们可以直接访问 this.drive()。正是由于存在标识符重叠，所以必须使用更加复杂的显式伪多态方法。

在支持相对多态的面向类的语言中，Car 和 Vehicle 之间的联系只在类定义的开头被创建，从而只需要在这一个地方维护两个类的联系。

但是在 JavaScript 中（由于屏蔽）使用显式伪多态会在所有需要使用（伪）多态引用的地方创建一个函数关联，这会极大地增加维护成本。此外，由于显式伪多态可以模拟多重继承，所以它会进一步增加代码的复杂度和维护难度。

使用伪多态通常会导致代码变得更加复杂、难以阅读并且难以维护，因此应当尽量避免使用显式伪多态，因为这样做往往得不偿失。

### 2. 混合复制

回顾一下之前提到的 mixin(..) 函数：

```
// 非常简单的 mixin(..) 例子 :
function mixin( sourceObj, targetObj ) {
```

```
    for (var key in sourceObj) {
        // 只会在不存在的情况下复制
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}
```

现在我们来分析一下 mixin(..) 的工作原理。它会遍历 sourceObj（本例中是 Vehicle）的属性，如果在 targetObj（本例中是 Car）没有这个属性就会进行复制。由于我们是在目标对象初始化之后才进行复制，因此一定要小心不要覆盖目标对象的原有属性。

如果我们是先进行复制然后对 Car 进行特殊化的话，就可以跳过存在性检查。不过这种方法并不好用并且效率更低，所以不如第一种方法常用：

```
// 另一种混入函数，可能有重写风险
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        targetObj[key] = sourceObj[key];
    }

    return targetObj;
}

var Vehicle = {
    // ...
};

// 首先创建一个空对象并把 Vehicle 的内容复制进去
var Car = mixin( Vehicle, { } );

// 然后把新内容复制到 Car 中
mixin( {
    wheels: 4,

    drive: function() {
        // ...
    }
}, Car );
```

这两种方法都可以把不重叠的内容从 Vehicle 中显性复制到 Car 中。“混入”这个名字来源于这个过程的另一种解释：Car 中混合了 Vehicle 的内容，就像你把巧克力片混合到你最喜欢的饼干面团中一样。

复制操作完成后，Car 就和 Vehicle 分离了，向 Car 中添加属性不会影响 Vehicle，反之亦然。

> 这里跳过了一些小细节，实际上，在复制完成之后两者之间仍然有一些巧妙的方法可以“影响”到对方，例如引用同一个对象（比如一个数组）。

由于两个对象引用的是同一个函数，因此这种复制（或者说混入）实际上并不能完全模拟面向类的语言中的复制。

JavaScript 中的函数无法（用标准、可靠的方法）真正地复制，所以你只能复制对共享函数对象的引用（函数就是对象；参见第 3 章）。如果你修改了共享的函数对象（比如 `ignition()`），比如添加了一个属性，那 `Vehicle` 和 `Car` 都会受到影响。

显式混入是 JavaScript 中一个很棒的机制，不过它的功能也没有看起来那么强大。虽然它可以把一个对象的属性复制到另一个对象中，但是这其实并不能带来太多的好处，无非就是少几条定义语句，而且还会带来我们刚才提到的函数对象引用问题。

如果你向目标对象中显式混入超过一个对象，就可以部分模仿多重继承行为，但是仍没有直接的方式来处理函数和属性的同名问题。有些开发者 / 库提出了“晚绑定”技术和其他的一些解决方法，但是从根本上来说，使用这些“诡计”通常会（降低性能并且）得不偿失。

一定要注意，只在能够提高代码可读性的前提下使用显式混入，避免使用增加代码理解难度或者让对象关系更加复杂的模式。

如果使用混入时感觉越来越困难，那或许你应该停止使用它了。实际上，如果你必须使用一个复杂的库或者函数来实现这些细节，那就标志着你的方法是有问题的或者是不必要的。第 6 章会试着提出一种更简单的方法，它能满足这些需求并且可以避免所有的问题。

### 3. 寄生继承

显式混入模式的一种变体被称为“寄生继承”，它既是显式的又是隐式的，主要推广者是 Douglas Crockford。

下面是它的工作原理：

```
//“传统的 JavaScript 类”Vehicle
function Vehicle() {
    this.engines = 1;
}
Vehicle.prototype.ignition = function() {
    console.log( "Turning on my engine." );
};
Vehicle.prototype.drive = function() {
    this.ignition();
    console.log( "Steering and moving forward!" );
};
```

```
//“寄生类”Car
function Car() {
    // 首先，car 是一个 Vehicle
    var car = new Vehicle();

    // 接着我们对 car 进行定制
    car.wheels = 4;

    // 保存到 Vehicle::drive() 的特殊引用
    var vehDrive = car.drive;

    // 重写 Vehicle::drive()
    car.drive = function() {
        vehDrive.call( this );
        console.log(
            "Rolling on all " + this.wheels + " wheels!"
        );

    return car;
}

var myCar = new Car();

myCar.drive();
// 发动引擎。
// 手握方向盘!
// 全速前进!
```

如你所见，首先我们复制一份 Vehicle 父类（对象）的定义，然后混入子类（对象）的定义（如果需要的话保留到父类的特殊引用），然后用这个复合对象构建实例。

> 调用 new Car() 时会创建一个新对象并绑定到 Car 的 this 上（参见第 2 章）。但是因为我们没有使用这个对象而是返回了我们自己的 car 对象，所以最初被创建的这个对象会被丢弃，因此可以不使用 new 关键字调用 Car()。这样做得到的结果是一样的，但是可以避免创建并丢弃多余的对象。

### 4.4.2 隐式混入

隐式混入和之前提到的显式伪多态很像，因此也具备同样的问题。

思考下面的代码：

```
var Something = {
    cool: function() {
        this.greeting = "Hello World";
        this.count = this.count ? this.count + 1 : 1;
    }
};
```

```
Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
    cool: function() {
        // 隐式把 Something 混入 Another
        Something.cool.call( this );
    }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1（count 不是共享状态）
```

通过在构造函数调用或者方法调用中使用 `Something.cool.call( this )`，我们实际上“借用”了函数 `Something.cool()` 并在 `Another` 的上下文中调用了它（通过 `this` 绑定；参加第 2 章）。最终的结果是 `Something.cool()` 中的赋值操作都会应用在 `Another` 对象上而不是 `Something` 对象上。

因此，我们把 `Something` 的行为“混入”到了 `Another` 中。

虽然这类技术利用了 `this` 的重新绑定功能，但是 `Something.cool.call( this )` 仍然无法变成相对（而且更灵活的）引用，所以使用时千万要小心。通常来说，尽量避免使用这样的结构，以保证代码的整洁和可维护性。

## 4.5 小结

类是一种设计模式。许多语言提供了对于面向类软件设计的原生语法。JavaScript 也有类似的语法，但是和其他语言中的类完全不同。

类意味着复制。

传统的类被实例化时，它的行为会被复制到实例中。类被继承时，行为也会被复制到子类中。

多态（在继承链的不同层次名称相同但是功能不同的函数）看起来似乎是从子类引用父类，但是本质上引用的其实是复制的结果。

JavaScript 并不会（像类那样）自动创建对象的副本。

混入模式（无论显式还是隐式）可以用来模拟类的复制行为，但是通常会产生丑陋并且脆弱的语法，比如显式伪多态（`OtherObj.methodName.call(this, ...)`），这会让代码更加难懂并且难以维护。

此外，显式混入实际上无法完全模拟类的复制行为，因为对象（和函数！别忘了函数也是对象）只能复制引用，无法复制被引用的对象或者函数本身。忽视这一点会导致许多问题。

总地来说，在 JavaScript 中模拟类是得不偿失的，虽然能解决当前的问题，但是可能会埋下更多的隐患。

# 第 5 章

# 原型

第 3 章和第 4 章多次提到了 [[Prototype]] 链，但没有说它到底是什么。现在我们来详细介绍一下它。

> 第 4 章中介绍的所有模拟类复制行为的方法，如各种混入，都没有使用 [[Prototype]] 链机制。

## 5.1 [[Prototype]]

JavaScript 中的对象有一个特殊的 [[Prototype]] 内置属性，其实就是对于其他对象的引用。几乎所有的对象在创建时 [[Prototype]] 属性都会被赋予一个非空的值。

注意：很快我们就可以看到，对象的 [[Prototype]] 链接可以为空，虽然很少见。

思考下面的代码：

```
var myObject = {
    a:2
};

myObject.a; // 2
```

[[Prototype]] 引用有什么用呢？在第 3 章中我们说过，当你试图引用对象的属性时会触发

[[Get]] 操作，比如 myObject.a。对于默认的 [[Get]] 操作来说，第一步是检查对象本身是否有这个属性，如果有的话就使用它。

> ES6 中的 Proxy 超出了本书的范围（但是在本系列之后的书中会介绍），但是要注意，如果包含 Proxy 的话，我们这里对 [[Get]] 和 [[Put]] 的讨论就不适用。

但是如果 a 不在 myObject 中，就需要使用对象的 [[Prototype]] 链了。

对于默认的 [[Get]] 操作来说，如果无法在对象本身找到需要的属性，就会继续访问对象的 [[Prototype]] 链：

```
var anotherObject = {
    a:2
};

// 创建一个关联到 anotherObject 的对象
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

> 稍后我们会介绍 Object.create(..) 的原理，现在只需要知道它会创建一个对象并把这个对象的 [[Prototype]] 关联到指定的对象。

现在 myObject 对象的 [[Prototype]] 关联到了 anotherObject。显然 myObject.a 并不存在，但是尽管如此，属性访问仍然成功地（在 anotherObject 中）找到了值 2。

但是，如果 anotherObject 中也找不到 a 并且 [[Prototype]] 链不为空的话，就会继续查找下去。

这个过程会持续到找到匹配的属性名或者查找完整条 [[Prototype]] 链。如果是后者的话，[[Get]] 操作的返回值是 undefined。

使用 for..in 遍历对象时原理和查找 [[Prototype]] 链类似，任何可以通过原型链访问到（并且是 enumerable，参见第 3 章）的属性都会被枚举。使用 in 操作符来检查属性在对象中是否存在时，同样会查找对象的整条原型链（无论属性是否可枚举）：

```
var anotherObject = {
    a:2
};
```

```
// 创建一个关联到 anotherObject 的对象
var myObject = Object.create( anotherObject );

for (var k in myObject) {
    console.log("found: " + k);
}
// found: a

("a" in myObject); // true
```

因此，当你通过各种语法进行属性查找时都会查找 [[Prototype]] 链，直到找到属性或者查找完整条原型链。

### 5.1.1 Object.prototype

但是到哪里是 [[Prototype]] 的“尽头”呢？

所有普通的 [[Prototype]] 链最终都会指向内置的 Object.prototype。由于所有的“普通”（内置，不是特定主机的扩展）对象都“源于”（或者说把 [[Prototype]] 链的顶端设置为）这个 Object.prototype 对象，所以它包含 JavaScript 中许多通用的功能。

有些功能你应该已经很熟悉了，比如说 .toString() 和 .valueOf()，第 3 章还介绍过 .hasOwnProperty(..)。稍后我们还会介绍 .isPrototypeOf(..)，这个你可能不太熟悉。

### 5.1.2 属性设置和屏蔽

第 3 章提到过，给一个对象设置属性并不仅仅是添加一个新属性或者修改已有的属性值。现在我们完整地讲解一下这个过程：

```
myObject.foo = "bar";
```

如果 myObject 对象中包含名为 foo 的普通数据访问属性，这条赋值语句只会修改已有的属性值。

如果 foo 不是直接存在于 myObject 中，[[Prototype]] 链就会被遍历，类似 [[Get]] 操作。如果原型链上找不到 foo，foo 就会被直接添加到 myObject 上。

然而，如果 foo 存在于原型链上层，赋值语句 myObject.foo = "bar" 的行为就会有些不同（而且可能很出人意料）。稍后我们会进行介绍。

如果属性名 foo 既出现在 myObject 中也出现在 myObject 的 [[Prototype]] 链上层，那么就会发生屏蔽。myObject 中包含的 foo 属性会屏蔽原型链上层的所有 foo 属性，因为 myObject.foo 总是会选择原型链中最底层的 foo 属性。

屏蔽比我们想象中更加复杂。下面我们分析一下如果 foo 不直接存在于 myObject 中而是存

在于原型链上层时 `myObject.foo = "bar"` 会出现的三种情况。

1. 如果在 `[[Prototype]]` 链上层存在名为 `foo` 的普通数据访问属性（参见第 3 章）并且没有被标记为只读（`writable:false`），那就会直接在 `myObject` 中添加一个名为 `foo` 的新属性，它是*屏蔽属性*。
2. 如果在 `[[Prototype]]` 链上层存在 `foo`，但是它被标记为只读（`writable:false`），那么无法修改已有属性或者在 `myObject` 上创建屏蔽属性。如果运行在严格模式下，代码会抛出一个错误。否则，这条赋值语句会被忽略。总之，不会发生屏蔽。
3. 如果在 `[[Prototype]]` 链上层存在 `foo` 并且它是一个 setter（参见第 3 章），那就一定会调用这个 setter。`foo` 不会被添加到（或者说屏蔽于）`myObject`，也不会重新定义 `foo` 这个 setter。

大多数开发者都认为如果向 `[[Prototype]]` 链上层已经存在的属性（`[[Put]]`）赋值，就一定会触发屏蔽，但是如你所见，三种情况中只有一种（第一种）是这样的。

如果你希望在第二种和第三种情况下也屏蔽 `foo`，那就不能使用 `=` 操作符来赋值，而是使用 `Object.defineProperty(..)`（参见第 3 章）来向 `myObject` 添加 `foo`。

> 第二种情况可能是最令人意外的，只读属性会阻止 `[[Prototype]]` 链下层隐式创建（屏蔽）同名属性。这样做主要是为了模拟类属性的继承。你可以把原型链上层的 `foo` 看作是父类中的属性，它会被 `myObject` 继承（复制），这样一来 `myObject` 中的 `foo` 属性也是只读，所以无法创建。但是一定要注意，实际上并不会发生类似的继承复制（参见第 4 章和第 5 章）。这看起来有点奇怪，`myObject` 对象竟然会因为其他对象中有一个只读 `foo` 就不能包含 `foo` 属性。更奇怪的是，这个限制只存在于 `=` 赋值中，使用 `Object.defineProperty(..)` 并不会受到影响。

如果需要对屏蔽方法进行委托的话就不得不使用丑陋的显式伪多态（参见第 4 章）。通常来说，使用屏蔽得不偿失，所以应当尽量避免使用。第 6 章会介绍另一种不使用屏蔽的更加简洁的设计模式。

有些情况下会隐式产生屏蔽，一定要当心。思考下面的代码：

```
var anotherObject = {
    a:2
};

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
myObject.a; // 2
```

```
anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // 隐式屏蔽!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true
```

尽管 myObject.a++ 看起来应该（通过委托）查找并增加 anotherObject.a 属性，但是别忘了 ++ 操作相当于 myObject.a = myObject.a + 1。因此 ++ 操作首先会通过 [[Prototype]] 查找属性 a 并从 anotherObject.a 获取当前属性值 2，然后给这个值加 1，接着用 [[Put]] 将值 3 赋给 myObject 中新建的屏蔽属性 a，天呐！

修改委托属性时一定要小心。如果想让 anotherObject.a 的值增加，唯一的办法是 anotherObject.a++。

## 5.2 "类"

现在你可能会很好奇：为什么一个对象需要关联到另一个对象？这样做有什么好处？这个问题非常好，但是在回答之前我们首先要理解 [[Prototype]]"不是"什么。

第 4 章中我们说过，JavaScript 和面向类的语言不同，它并没有类来作为对象的抽象模式或者说蓝图。JavaScript 中只有对象。

实际上，JavaScript 才是真正应该被称为"面向对象"的语言，因为它是少有的可以不通过类，直接创建对象的语言。

在 JavaScript 中，类无法描述对象的行，（因为根本就不存在类！）对象直接定义自己的行为。再说一遍，JavaScript 中只有对象。

### 5.2.1 "类"函数

多年以来，JavaScript 中有一种奇怪的行为一直在被无耻地滥用，那就是模仿类。我们会仔细分析这种方法。

这种奇怪的"类似类"的行为利用了函数的一种特殊特性：所有的函数默认都会拥有一个名为 prototype 的公有并且不可枚举（参见第 3 章）的属性，它会指向另一个对象：

```
function Foo() {
    // ...
}

Foo.prototype; // { }
```

这个对象通常被称为 Foo 的原型，因为我们通过名为 Foo.prototype 的属性引用来访问它。然而不幸的是，这个术语对我们造成了极大的误导，稍后我们就会看到。如果是我的话就会叫它“之前被称为 Foo 的原型的那个对象”。好吧我是开玩笑的，你觉得“被贴上‘Foo 点 prototype’标签的对象”这个名字怎么样？

抛开名字不谈，这个对象到底是什么？

最直接的解释就是，这个对象是在调用 new Foo()（参见第 2 章）时创建的，最后会被（有点武断地）关联到这个“Foo 点 prototype”对象上。

我们来验证一下：

```
function Foo() {
    // ...
}

var a = new Foo();

Object.getPrototypeOf( a ) === Foo.prototype; // true
```

调用 new Foo() 时会创建 a（具体的 4 个步骤参见第 2 章），其中的一步就是给 a 一个内部的 [[Prototype]] 链接，关联到 Foo.prototype 指向的那个对象。

暂停一下，仔细思考这条语句的含义。

在面向类的语言中，类可以被复制（或者说实例化）多次，就像用模具制作东西一样。我们在第 4 章中看到过，之所以会这样是因为实例化（或者继承）一个类就意味着“把类的行为复制到物理对象中”，对于每一个新实例来说都会重复这个过程。

但是在 JavaScript 中，并没有类似的复制机制。你不能创建一个类的多个实例，只能创建多个对象，它们 [[Prototype]] 关联的是同一个对象。但是在默认情况下并不会进行复制，因此这些对象之间并不会完全失去联系，它们是互相关联的。

new Foo() 会生成一个新对象（我们称之为 a），这个新对象的内部链接 [[Prototype]] 关联的是 Foo.prototype 对象。

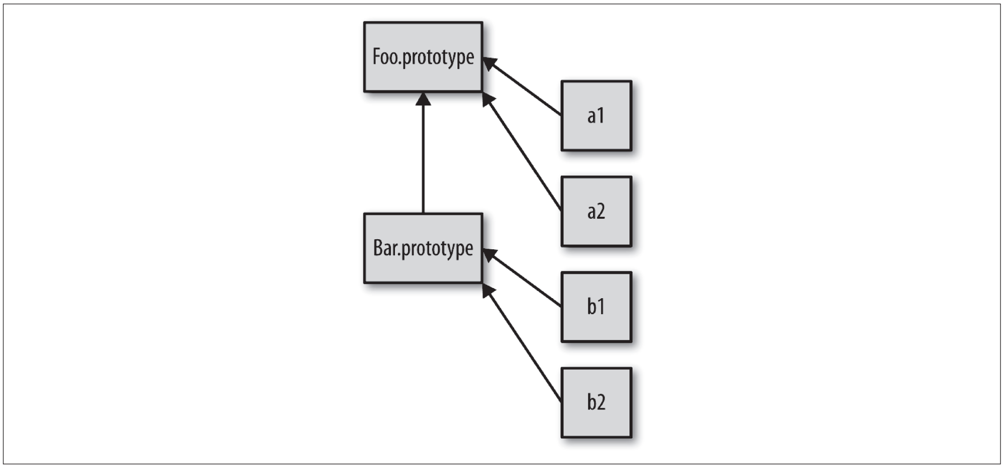
最后我们得到了两个对象，它们之间互相关联，就是这样。我们并没有初始化一个类，实际上我们并没有从“类”中复制任何行为到一个对象中，只是让两个对象互相关联。

实际上，绝大多数 JavaScript 开发者不知道的秘密是，new Foo() 这个函数调用实际上并没有直接创建关联，这个关联只是一个意外的副作用。new Foo() 只是间接完成了我们的目标：一个关联到其他对象的新对象。

那么有没有更直接的方法来做到这一点呢？当然！功臣就是 Object.create(..)，不过我们现在暂时不介绍它。

### 关于名称

在 JavaScript 中，我们并不会将一个对象（“类”）复制到另一个对象（“实例”），只是将它们关联起来。从视觉角度来说，`[[Prototype]]` 机制如下图所示，箭头从右到左，从下到上：




这个机制通常被称为*原型继承*（稍后我们会分析具体代码），它常常被视为动态语言版本的类继承。这个名称主要是为了对应面向类的世界中“继承”的意义，但是违背（写作违背，读作推翻）了动态脚本中对应的语义。

“继承”这个词会让人产生非常强的心理预期（参见第 4 章）。仅仅在前面加上“原型”并不能区分出 JavaScript 中和类继承*几乎完全相反*的行为，因此在过去 20 年中造成了极大的误解。

在我看来，在“继承”前面加上“原型”对于事实的曲解就好像一只手拿橘子一只手拿苹果然后把苹果叫作“红橘子”一样。无论添加什么标签都无法改变*事实*：一种水果是苹果，另一种是橘子。

更好的方法是直接把苹果叫作苹果——使用更加准确并且直接的术语。这样有助于理解它们的相似之处以及不同之处，因为我们大家都明白“苹果”的含义。

因此我认为这个容易混淆的组合术语“原型继承”（以及使用其他面向类的术语比如“类”、“构造函数”、“实例”、“多态”，等等）严重影响了大家对于 JavaScript 机制真实原理的理解。

*继承*意味着复制操作，JavaScript（默认）并不会复制对象属性。相反，JavaScript 会在两个对象之间创建一个关联，这样一个对象就可以通过*委托*访问另一个对象的属性和函数。*委托*（参见第 6 章）这个术语可以更加准确地描述 JavaScript 中对象的关联机制。

还有个偶尔会用到的 JavaScript 术语*差异继承*。基本原则是在描述对象行为时，使用其不

同于普遍描述的特质。举例来说，描述汽车时你会说汽车是有四个轮子的一种交通工具，但是你不会重复描述交通工具具备的通用特性（比如引擎）。

如果你把 JavaScript 中对象的所有委托行为都归结到对象本身并且把对象看作是实物的话，那就（差不多）可以理解差异继承了。

但是和原型继承一样，差异继承会更多是你脑中构建出的模型，而非真实情况。它忽略了一个事实，那就是对象 B 实际上并不是被差异构造出来的，我们只是定义了 B 的一些指定特性，其他没有定义的东西都变成了“洞”。而这些洞（或者说缺少定义的空白处）最终会被委托行为“填满”。

默认情况下，对象并不会像差异继承暗示的那样通过复制生成。因此，差异继承也不适合用来描述 JavaScript 的 [[Prototype]] 机制。

当然，如果你喜欢，完全可以使用差异继承这个术语，但是无论如何它只适用于你脑中的模型，并不符合引擎的真实行为。

### 5.2.2 “构造函数”

好了，回到之前的代码：

```
function Foo() {
    // ...
}

var a = new Foo();
```

到底是什么让我们认为 Foo 是一个“类”呢？

其中一个原因是我们看到了关键字 new，在面向类的语言中构造类实例时也会用到它。另一个原因是，看起来我们执行了类的构造函数方法，Foo() 的调用方式很像初始化类时类构造函数的调用方式。

除了令人迷惑的“构造函数”语义外，Foo.prototype 还有另一个绝招。思考下面的代码：

```
function Foo() {
    // ...
}

Foo.prototype.constructor === Foo; // true

var a = new Foo();
a.constructor === Foo; // true
```

Foo.prototype 默认（在代码中第一行声明时！）有一个公有并且不可枚举（参见第 3 章）的属性 .constructor，这个属性引用的是对象关联的函数（本例中是 Foo）。此外，我们

可以看到通过“构造函数”调用 new Foo() 创建的对象也有一个 .constructor 属性，指向“创建这个对象的函数”。

> 实际上 a 本身并没有 .constructor 属性。而且，虽然 a.constructor 确实指向 Foo 函数，但是这个属性并不是表示 a 由 Foo“构造”，稍后我们会解释。

哦耶，好吧……按照 JavaScript 世界的惯例，“类”名首字母要大写，所以名字写作 Foo 而非 foo 似乎也提示它是一个“类”。显而易见，是吧 ?!

> 这个惯例影响力非常大，以至于如果你用 new 来调用小写方法或者不用 new 调用首字母大写的函数，许多 JavaScript 开发者都会责怪你。这很令人吃惊，我们竟然会如此努力地维护 JavaScript 中（假）“面向类”的权力，尽管对于 JavaScript 引擎来说首字母大写没有任何意义。

### 1. 构造函数还是调用

上一段代码很容易让人认为 Foo 是一个构造函数，因为我们使用 new 来调用它并且看到它“构造”了一个对象。

实际上，Foo 和你程序中的其他函数没有任何区别。函数本身并不是构造函数，然而，当你在普通的函数调用前面加上 new 关键字之后，就会把这个函数调用变成一个“构造函数调用”。实际上，new 会劫持所有普通函数并用构造对象的形式来调用它。

举例来说：

```
function NothingSpecial() {
    console.log( "Don't mind me!" );
}

var a = new NothingSpecial();
// "Don't mind me!"

a; // {}
```

NothingSpecial 只是一个普通的函数，但是使用 new 调用时，它就会构造一个对象并赋值给 a，这看起来像是 new 的一个副作用（无论如何都会构造一个对象）。这个调用是一个构造函数调用，但是 NothingSpecial 本身并不是一个构造函数。

换句话说，在 JavaScript 中对于“构造函数”最准确的解释是，所有带 new 的函数调用。

函数不是构造函数，但是当且仅当使用 new 时，函数调用会变成“构造函数调用”。

### 5.2.3 技术

我们是不是已经介绍了 JavaScript 中所有和“类”相关的问题了呢?

不是。JavaScript 开发者绞尽脑汁想要模仿类的行为:

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

var a = new Foo( "a" );
var b = new Foo( "b" );

a.myName(); // "a"
b.myName(); // "b"
```

这段代码展示了另外两种“面向类”的技巧:

1. `this.name = name` 给每个对象(也就是 `a` 和 `b`,参见第 2 章中的 `this` 绑定)都添加了 `.name` 属性,有点像类实例封装的数据值。
2. `Foo.prototype.myName = ...` 可能个更有趣的技巧,它会给 `Foo.prototype` 对象添加一个属性(函数)。现在,`a.myName()` 可以正常工作,但是你可能会觉得很惊讶,这是什么原理呢?

在这段代码中,看起来似乎创建 `a` 和 `b` 时会把 `Foo.prototype` 对象复制到这两个对象中,然而事实并不是这样。

在本章开头介绍默认 `[[Get]]` 算法时我们介绍过 `[[Prototype]]` 链,以及当属性不直接存在于对象中时如何通过它来进行查找。

因此,在创建的过程中,`a` 和 `b` 的内部 `[[Prototype]]` 都会关联到 `Foo.prototype` 上。当 `a` 和 `b` 中无法找到 `myName` 时,它会(通过委托,参见第 6 章)在 `Foo.prototype` 上找到。

#### 回顾“构造函数”

之前讨论 `.constructor` 属性时我们说过,看起来 `a.constructor === Foo` 为真意味着 `a` 确实有一个指向 `Foo` 的 `.constructor` 属性,但是事实不是这样。

这是一个很不幸的误解。实际上,`.constructor` 引用同样被委托给了 `Foo.prototype`,而 `Foo.prototype.constructor` 默认指向 `Foo`。

把 `.constructor` 属性指向 `Foo` 看作是 a 对象由 Foo“构造”非常容易理解,但这只不过是一种虚假的安全感。`a.constructor` 只是通过默认的 `[[Prototype]]` 委托指向 `Foo`,这和

“构造”毫无关系。相反，对于 .constructor 的错误理解很容易对你自己产生误导。

举例来说，Foo.prototype 的 .constructor 属性只是 Foo 函数在声明时的默认属性。如果你创建了一个新对象并替换了函数默认的 .prototype 对象引用，那么新对象并不会自动获得 .constructor 属性。

思考下面的代码：

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // 创建一个新原型对象

var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

Object(..) 并没有“构造”a1，对吧？看起来应该是 Foo()“构造”了它。大部分开发者都认为是 Foo() 执行了构造工作，但是问题在于，如果你认为“constructor”表示“由……构造”的话，a1.constructor 应该是 Foo，但是它并不是 Foo！

到底怎么回事？ a1 并没有 .constructor 属性，所以它会委托 [[Prototype]] 链上的 Foo.prototype。但是这个对象也没有 .constructor 属性（不过默认的 Foo.prototype 对象有这个属性！），所以它会继续委托，这次会委托给委托链顶端的 Object.prototype。这个对象有 .constructor 属性，指向内置的 Object(..) 函数。

错误观点已被摧毁。

当然，你可以给 Foo.prototype 添加一个 .constructor 属性，不过这需要手动添加一个符合正常行为的不可枚举（参见第 3 章）属性。

举例来说：

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // 创建一个新原型对象

// 需要在 Foo.prototype 上“修复”丢失的 .constructor 属性
// 新对象属性起到 Foo.prototype 的作用
// 关于 defineProperty(..)，参见第 3 章
Object.defineProperty( Foo.prototype, "constructor" , {
    enumerable: false,
    writable: true,
    configurable: true,
    value: Foo // 让 .constructor 指向 Foo
} );
```

修复 .constructor 需要很多手动操作。所有这些工作都是源于把“constructor”错误地理解为“由……构造”，这个误解的代价实在太高了。

实际上，对象的 .constructor 会默认指向一个函数，这个函数可以通过对象的 .prototype 引用。“constructor”和“prototype”这两个词本身的含义可能适用也可能不适用。最好的办法是记住这一点“constructor 并不表示被构造”。

.constructor 并不是一个不可变属性。它是不可枚举（参见上面的代码）的，但是它的值是可写的（可以被修改）。此外，你可以给任意 [[Prototype]] 链中的任意对象添加一个名为 constructor 的属性或者对其进行修改，你可以任意对其赋值。

和 [[Get]] 算法查找 [[Prototype]] 链的机制一样，.constructor 属性引用的目标可能和你想的完全不同。

现在你应该明白这个属性多么随意了吧？

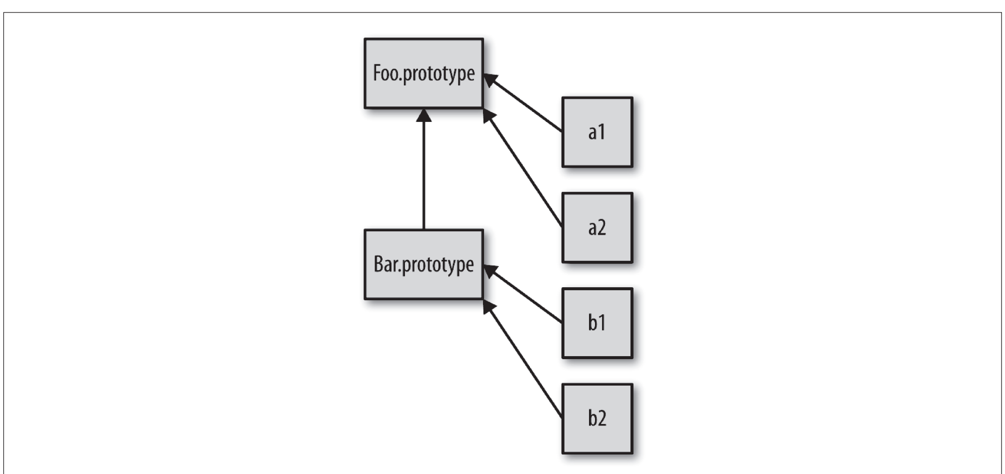
结论？一些随意的对象属性引用，比如 a1.constructor，实际上是不被信任的，它们不一定会指向默认的函数引用。此外，很快我们就会看到，稍不留神 a1.constructor 就可能会指向你意想不到的地方。

a1.constructor 是一个非常不可靠并且不安全的引用。通常来说要尽量避免使用这些引用。

## 5.3 （原型）继承

我们已经看过了许多 JavaScript 程序中常用的模拟类行为的方法，但是如果没有“继承”机制的话，JavaScript 中的类就只是一个空架子。

实际上，我们已经了解了通常被称作原型继承的机制，a 可以“继承”Foo.prototype 并访问 Foo.prototype 的 myName() 函数。但是之前我们只把继承看作是类和类之间的关系，并没有把它看作是类和实例之间的关系：



还记得这张图吗，它不仅展示出对象（实例）a1 到 Foo.prototype 的委托关系，还展示出 Bar.prototype 到 Foo.prototype 的委托关系，而后者和类继承很相似，只有箭头的方向不同。图中由下到上的箭头表明这是委托关联，不是复制操作。

下面这段代码使用的就是典型的“原型风格”：

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name,label) {
    Foo.call( this, name );
    this.label = label;
}

// 我们创建了一个新的 Bar.prototype 对象并关联到 Foo.prototype
Bar.prototype = Object.create( Foo.prototype );

// 注意！现在没有 Bar.prototype.constructor 了
// 如果你需要这个属性的话可能需要手动修复一下它

Bar.prototype.myLabel = function() {
    return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
```

如果不明白为什么 this 指向 a 的话，请查看第 2 章。

这段代码的核心部分就是语句 Bar.prototype = Object.create( Foo.prototype )。调用 Object.create(..) 会凭空创建一个“新”对象并把新对象内部的 [[Prototype]] 关联到你指定的对象（本例中是 Foo.prototype）。

换句话说，这条语句的意思是：“创建一个新的 Bar.prototype 对象并把它关联到 Foo.prototype”。

声明 function Bar() { .. } 时，和其他函数一样，Bar 会有一个 .prototype 关联到默认的对象，但是这个对象并不是我们想要的 Foo.prototype。因此我们创建了一个新对象并把

它关联到我们希望的对象上，直接把原始的关联对象抛弃掉。

注意，下面这两种方式是常见的错误做法，实际上它们都存在一些问题：

```
// 和你想要的机制不一样!
Bar.prototype = Foo.prototype;

// 基本上满足你的需求，但是可能会产生一些副作用 :(
Bar.prototype = new Foo();
```

`Bar.prototype = Foo.prototype` 并不会创建一个关联到 `Bar.prototype` 的新对象，它只是让 `Bar.prototype` 直接引用 `Foo.prototype` 对象。因此当你执行类似 `Bar.prototype.myLabel = ...` 的赋值语句时会直接修改 `Foo.prototype` 对象本身。显然这不是你想要的结果，否则你根本不需要 `Bar` 对象，直接使用 `Foo` 就可以了，这样代码也会更简单一些。

`Bar.prototype = new Foo()` 的确会创建一个关联到 `Bar.prototype` 的新对象。但是它使用了 `Foo(..)` 的"构造函数调用"，如果函数 `Foo` 有一些副作用（比如写日志、修改状态、注册到其他对象、给 `this` 添加数据属性，等等）的话，就会影响到 `Bar()` 的"后代"，后果不堪设想。

因此，要创建一个合适的关联对象，我们必须使用 `Object.create(..)` 而不是使用具有副作用的 `Foo(..)`。这样做唯一的缺点就是需要创建一个新对象然后把旧对象抛弃掉，不能直接修改已有的默认对象。

如果能有一个标准并且可靠的方法来修改对象的 `[[Prototype]]` 关联就好了。在 ES6 之前，我们只能通过设置 `.__proto__` 属性来实现，但是这个方法并不是标准并且无法兼容所有浏览器。ES6 添加了辅助函数 `Object.setPrototypeOf(..)`，可以用标准并且可靠的方法来修改关联。

我们来对比一下两种把 `Bar.prototype` 关联到 `Foo.prototype` 的方法：

```
// ES6 之前需要抛弃默认的 Bar.prototype
Bar.ptototype = Object.create( Foo.prototype );

// ES6 开始可以直接修改现有的 Bar.prototype
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

如果忽略掉 `Object.create(..)` 方法带来的轻微性能损失（抛弃的对象需要进行垃圾回收），它实际上比 ES6 及其之后的方法更短而且可读性更高。不过无论如何，这是两种完全不同的语法。

## 检查"类"关系

假设有对象 `a`，如何寻找对象 `a` 委托的对象（如果存在的话）呢？在传统的面向类环境中，

检查一个实例（JavaScript 中的对象）的继承祖先（JavaScript 中的委托关联）通常被称为内省（或者反射）。

思考下面的代码：

```
function Foo() {
    // ...
}

Foo.prototype.blah = ...;

var a = new Foo();
```

我们如何通过内省找出 a 的“祖先”（委托关联）呢？第一种方法是站在“类”的角度来判断：

```
a instanceof Foo; // true
```

instanceof 操作符的左操作数是一个普通的对象，右操作数是一个函数。instanceof 回答的问题是：在 a 的整条 [[Prototype]] 链中是否有指向 Foo.prototype 的对象？

可惜，这个方法只能处理对象（a）和函数（带 .prototype 引用的 Foo）之间的关系。如果你想判断两个对象（比如 a 和 b）之间是否通过 [[Prototype]] 链关联，只用 instanceof 无法实现。

> 如果使用内置的 .bind(..) 函数来生成一个硬绑定函数（参见第 2 章）的话，该函数是没有 .prototype 属性的。在这样的函数上使用 instanceof 的话，目标函数的 .prototype 会代替硬绑定函数的 .prototype。
>
> 通常我们不会在“构造函数调用”中使用硬绑定函数，不过如果你这么做的话，实际上相当于直接调用目标函数。同理，在硬绑定函数上使用 instanceof 也相当于直接在目标函数上使用 instanceof。

下面这段荒谬的代码试图站在“类”的角度使用 instanceof 来判断两个对象的关系：

```
// 用来判断 o1 是否关联到（委托）o2 的辅助函数
function isRelatedTo(o1, o2) {
    function F(){}
    F.prototype = o2;
    return o1 instanceof F;
}

var a = {};
var b = Object.create( a );

isRelatedTo( b, a ); // true
```

在 isRelatedTo(..) 内部我们声明了一个一次性函数 F，把它的 .prototype 重新赋值并指向对象 o2，然后判断 o1 是否是 F 的一个“实例”。显而易见，o1 实际上并没有继承 F 也不是由 F 构造，所以这种方法非常愚蠢并且容易造成误解。问题的关键在于思考的角度，强行在 JavaScript 中应用类的语义（在本例中就是使用 instanceof）就会造成这种尴尬的局面。

下面是第二种判断 [[Prototype]] 反射的方法，它更加简洁：

```
Foo.prototype.isPrototypeOf( a ); // true
```

注意，在本例中，我们实际上并不关心（甚至不需要）Foo，我们只需要一个可以用来判断的对象（本例中是 Foo.prototype）就行。isPrototypeOf(..) 回答的问题是：在 a 的整条 [[Prototype]] 链中是否出现过 Foo.prototype ？

同样的问题，同样的答案，但是在第二种方法中并不需要间接引用函数（Foo），它的 .prototype 属性会被自动访问。

我们只需要两个对象就可以判断它们之间的关系。举例来说：

```
// 非常简单：b 是否出现在 c 的 [[Prototype]] 链中？
b.isPrototypeOf( c );
```

注意，这个方法并不需要使用函数（“类”），它直接使用 b 和 c 之间的对象引用来判断它们的关系。换句话说，语言内置的 isPrototypeOf(..) 函数就是我们的 isRelatedTo(..) 函数。

我们也可以直接获取一个对象的 [[Prototype]] 链。在 ES5 中，标准的方法是：

```
Object.getPrototypeOf( a );
```

可以验证一下，这个对象引用是否和我们想的一样：

```
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

绝大多数（不是所有！）浏览器也支持一种非标准的方法来访问内部 [[Prototype]] 属性：

```
a.__proto__ === Foo.prototype; // true
```

这个奇怪的 .__proto__（在 ES6 之前并不是标准！）属性“神奇地”引用了内部的 [[Prototype]] 对象，如果你想直接查找（甚至可以通过 .__proto__.__ptoto__... 来遍历）原型链的话，这个方法非常有用。

和我们之前说过的 .constructor 一样，.__proto__ 实际上并不存在于你正在使用的对象中（本例中是 a）。实际上，它和其他的常用函数（.toString()、.isPrototypeOf(..)，等等）

一样，存在于内置的 Object.prototype 中。（它们是不可枚举的，参见第 2 章。）

此外，.__proto__ 看起来很像一个属性，但是实际上它更像一个 getter/setter（参见第 3 章）。

.__proto__ 的实现大致上是这样的（对象属性的定义参见第 3 章）：

```
Object.defineProperty( Object.prototype, "__proto__", {
    get: function() {
        return Object.getPrototypeOf( this );
    },
    set: function(o) {
        // ES6 中的 setPrototypeOf(..)
        Object.setPrototypeOf( this, o );
        return o;
    }
} );
```

因此，访问（获取值）a.__proto__ 时，实际上是调用了 a.__proto__()（调用 getter 函数）。虽然 getter 函数存在于 Object.prototype 对象中，但是它的 this 指向对象 a（this 的绑定规则参见第 2 章），所以和 Object.getPrototypeOf( a ) 结果相同。

.__proto__ 是可设置属性，之前的代码中使用 ES6 的 Object.setPrototypeOf(..) 进行设置。然而，通常来说你不需要修改已有对象的 [[Prototype]]。

一些框架会使用非常复杂和高端的技术来实现"子类"机制，但是通常来说，我们不推荐这种用法，因为这会极大地增加代码的阅读难度和维护难度。

> ES6 中的 class 关键字可以在内置的类型（比如 Array）上实现类似"子类"的功能。详情参考附录 A 中关于 ES6 中 class 语法的介绍。

我们只有在一些特殊情况下（我们前面讨论过）需要设置函数默认 .prototype 对象的 [[Prototype]]，让它引用其他对象（除了 Object.prototype）。这样可以避免使用全新的对象替换默认对象。此外，最好把 [[Prototype]] 对象关联看作是只读特性，从而增加代码的可读性。

> JavaScript 社区中对于双下划线有一个非官方的称呼，他们会把类似 __proto__ 的属性称为"笨蛋（dunder）"。所以，JavaScript 潮人会把 __proto__ 叫作"笨蛋 proto"。

# 5.4　对象关联

现在我们知道了，[[Prototype]] 机制就是存在于对象中的一个内部链接，它会引用其他对象。

通常来说，这个链接的作用是：如果在对象上没有找到需要的属性或者方法引用，引擎就会继续在 [[Prototype]] 关联的对象上进行查找。同理，如果在后者中也没有找到需要的引用就会继续查找它的 [[Prototype]]，以此类推。这一系列对象的链接被称为“原型链”。

## 5.4.1　创建关联

我们已经明白了为什么 JavaScript 的 [[Prototype]] 机制和类不一样，也明白了它如何建立对象间的关联。

那 [[Prototype]] 机制的意义是什么呢？为什么 JavaScript 开发者费这么大的力气（模拟类）在代码中创建这些关联呢？

还记得吗，本章前面曾经说过 Object.create(..) 是一个大英雄，现在是时候来弄明白为什么了：

```
var foo = {
    something: function() {
        console.log( "Tell me something good..." );
    }
};

var bar = Object.create( foo );

bar.something(); // Tell me something good...
```

Object.create(..) 会创建一个新对象（bar）并把它关联到我们指定的对象（foo），这样我们就可以充分发挥 [[Prototype]] 机制的威力（委托）并且避免不必要的麻烦（比如使用 new 的构造函数调用会生成 .prototype 和 .constructor 引用）。

> Object.create(null) 会创建一个拥有空（或者说 null）[[Prototype]] 链接的对象，这个对象无法进行委托。由于这个对象没有原型链，所以 instanceof 操作符（之前解释过）无法进行判断，因此总是会返回 false。这些特殊的空 [[Prototype]] 对象通常被称作“字典”，它们完全不会受到原型链的干扰，因此非常适合用来存储数据。

我们并不需要类来创建两个对象之间的关系，只需要通过委托来关联对象就足够了。而 Object.create(..) 不包含任何“类的诡计”，所以它可以完美地创建我们想要的关联关系。

### Object.create()的polyfill代码

Object.create(..) 是在 ES5 中新增的函数，所以在 ES5 之前的环境中（比如旧 IE）如果要支持这个功能的话就需要使用一段简单的 polyfill 代码，它部分实现了 Object.create(..) 的功能：

```
if (!Object.create) {
    Object.create = function(o) {
        function F(){}
        F.prototype = o;
        return new F();
    };
}
```

这段 polyfill 代码使用了一个一次性函数 F，我们通过改写它的 .prototype 属性使其指向想要关联的对象，然后再使用 new F() 来构造一个新对象进行关联。

由于 Object.create(..c) 可以被模拟，因此这个函数被应用得非常广泛。标准 ES5 中内置的 Object.create(..) 函数还提供了一系列附加功能，但是 ES5 之前的版本不支持这些功能。通常来说，这些功能的应用范围要小得多，但是出于完整性考虑，我们还是介绍一下：

```
var anotherObject = {
    a:2
};

var myObject = Object.create( anotherObject, {
    b: {
        enumerable: false,
        writable: true,
        configurable: false,
        value: 3
    },
    c: {
        enumerable: true,
        writable: false,
        configurable: false,
        value: 4
    }
});

myObject.hasOwnProperty( "a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true

myObject.a; // 2
myObject.b; // 3
myObject.c; // 4
```

Object.create(..) 的第二个参数指定了需要添加到新对象中的属性名以及这些属性的属性

描述符（参见第 3 章）。因为 ES5 之前的版本无法模拟属性操作符，所以 polyfill 代码无法实现这个附加功能。

通常来说并不会使用 `Object.create(..)` 的附加功能，所以对于大多数开发者来说，上面那段 polyfill 代码就足够了。

有些开发者更加严谨，他们认为只有能被完全模拟的函数才应该使用 polyfill 代码。由于 `Object.create(..)` 是只能部分模拟的函数之一，所以这些狭隘的人认为如果你需要在 ES5 之前的环境中使用 `Object.create(..)` 的特性，那不要使用 polyfill 代码，而是使用一个自定义函数并且名字不能是 `Object.create`。你可以把你自己的函数定义成这样：

```
function createAndLinkObject(o) {
    function F(){}
    F.prototype = o;
    return new F();
}

var anotherObject = {
    a:2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2
```

我并不赞同这个严格的观点，相反，我很赞同在 ES5 中使用上面那段 polyfill 代码。如何选择取决于你。

### 5.4.2　关联关系是备用

看起来对象之间的关联关系是处理“缺失”属性或者方法时的一种备用选项。这个说法有点道理，但是我认为这并不是 `[[Prototype]]` 的本质。

思考下面的代码：

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // "cool!"
```

由于存在 `[[Prototype]]` 机制，这段代码可以正常工作。但是如果你这样写只是为了让 `myObject` 在无法处理属性或者方法时可以使用备用的 `anotherObject`，那么你的软件就会

变得有点“神奇”，而且很难理解和维护。

这并不是说任何情况下都不应该选择备用这种设计模式，但是这在 JavaScript 中并不是很常见。所以如果你使用的是这种模式，那或许应当退后一步并重新思考一下这种模式是否合适。

> 在 ES6 中有一个被称为“代理”（Proxy）的高端功能，它实现的就是“方法无法找到”时的行为。代理超出了本书的讨论范围，但是在本系列之后的书中会介绍。

*千万不要忽略这个微妙但是非常重要的区别。*

当你给开发者设计软件时，假设要调用 `myObject.cool()`，如果 `myObject` 中不存在 `cool()` 时这条语句也可以正常工作的话，那你的 API 设计就会变得很“神奇”，对于未来维护你软件的开发者来说这可能不太好理解。

但是你可以让你的 API 设计不那么“神奇”，同时仍然能发挥 `[[Prototype]]` 关联的威力：

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
    this.cool(); // 内部委托!
};

myObject.doCool(); // "cool!"
```

这里我们调用的 `myObject.doCool()` 是*实际存在于* `myObject` 中的，这可以让我们的 API 设计更加清晰（不那么“神奇”）。从内部来说，我们的实现遵循的是*委托设计模式*（参见第 6 章），通过 `[[Prototype]]` 委托到 `anotherObject.cool()`。

换句话说，内部委托比起直接委托可以让 API 接口设计更加清晰。下一章我们会详细解释委托。

## 5.5 小结

如果要访问对象中并不存在的一个属性，`[[Get]]` 操作（参见第 3 章）就会查找对象内部 `[[Prototype]]` 关联的对象。这个关联关系实际上定义了一条“原型链”（有点像嵌套的作

用域链)，在查找属性时会对它进行遍历。

所有普通对象都有内置的 `Object.prototype`，指向原型链的顶端（比如说全局作用域），如果在原型链中找不到指定的属性就会停止。`toString()`、`valueOf()` 和其他一些通用的功能都存在于 `Object.prototype` 对象上，因此语言中所有的对象都可以使用它们。

关联两个对象最常用的方法是使用 `new` 关键词进行函数调用，在调用的 4 个步骤（第 2 章）中会创建一个关联其他对象的新对象。

使用 `new` 调用函数时会把新对象的 `.prototype` 属性关联到“其他对象”。带 `new` 的函数调用通常被称为“构造函数调用”，尽管它们实际上和传统面向类语言中的*类构造函数*不一样。

虽然这些 JavaScript 机制和传统面向类语言中的“类初始化”和“类继承”很相似，但是 JavaScript 中的机制有一个核心区别，那就是不会进行复制，对象之间是通过内部的 `[[Prototype]]` 链关联的。

出于各种原因，以“继承”结尾的术语（包括“原型继承”）和其他面向对象的术语都无法帮助你理解 JavaScript 的*真实*机制（不仅仅是限制我们的思维模式）。

相比之下，“委托”是一个更合适的术语，因为对象之间的关系不是*复制*而是委托。

# 第 6 章
# 行为委托

第 5 章详细介绍了 [[Prototype]] 机制并说明了为什么在“类”或者“继承”的背景下讨论 [[Prototype]] 容易产生误解（这种不恰当的方式已经持续了 20 年）。我们搞清楚了繁杂的语法（各种 .prototype 代码），也见识了各种各样的陷阱（比如出人意料的 .constructor 和丑陋的伪多态语法），我们还看到了用来解决这些问题的各种“混入”方法。

你可能会很好奇，为什么看起来简单的事情会变得这么复杂。现在我们会把帘子拉开，看看后面到底有什么。不出意外，绝大多数 JavaScript 开发者从来没有如此深入地了解过 JavaScript，他们只是把这些交给一个“类”库来处理。

现在，我希望你不仅满足于掩盖这些细节并把它们交给一个“黑盒”库。忘掉令人困惑的类，我们用一种*更加简单直接*的方法来深入发掘一下 JavaScript 中对象的 [[Prototype]] 机制到底是什么。

首先简单回顾一下第 5 章的结论：[[Prototype]] 机制就是指对象中的一个内部链接引用另一个对象。

如果在第一个对象上没有找到需要的属性或者方法引用，引擎就会继续在 [[Prototype]] 关联的对象上进行查找。同理，如果在后者中也没有找到需要的引用就会继续查找它的 [[Prototype]]，以此类推。这一系列对象的链接被称为“原型链”。

换句话说，JavaScript 中这个机制的本质就是*对象之间的关联关系*。

这个观点对于理解本章的内容来说是非常基础并且非常重要的。

## 6.1 面向委托的设计

为了更好地学习如何更直观地使用 [[Prototype]]，我们必须认识到它代表的是一种不同于类（参见第 4 章）的设计模式。

> 面向类的设计中有些原则依然有效，因此不要把所有知识都抛掉。（只需要抛掉大部分就够了！）举例来说，封装是非常有用的，它同样可以应用在委托中（虽然不太常见）。

我们需要试着把思路从类和继承的设计模式转换到委托行为的设计模式。如果你在学习或者工作的过程中几乎一直在使用类，那转换思路可能不太自然并且不太舒服。你可能需要多重复几次才能熟悉这种思维模式。

首先我会带你们进行一些理论训练，然后再传授一些能够应用在代码中的具体实例。

### 6.1.1 类理论

假设我们需要在软件中建模一些类似的任务（“XYZ”、“ABC”等）。

如果使用类，那设计方法可能是这样的：定义一个通用父（基）类，可以将其命名为 Task，在 Task 类中定义所有任务都有的行为。接着定义子类 XYZ 和 ABC，它们都继承自 Task 并且会添加一些特殊的行为来处理对应的任务。

非常重要的是，类设计模式鼓励你在继承时使用方法重写（和多态），比如说在 XYZ 任务中重写 Task 中定义的一些通用方法，甚至在添加新行为时通过 super 调用这个方法的原始版本。你会发现许多行为可以先“抽象”到父类然后再用子类进行特殊化（重写）。

下面是对应的伪代码：

```
class Task {
    id;

    // 构造函数 Task()
    Task(ID) { id = ID; }
    outputTask() { output( id ); }
}

class XYZ inherits Task {
    label;

    // 构造函数 XYZ()
    XYZ(ID,Label) { super( ID ); label = Label; }
    outputTask() { super(); output( label ); }
}
```

```
class ABC inherits Task {
    // ...
}
```

现在你可以实例化子类 XYZ 的一些副本然后使用这些实例来执行任务“XYZ”。这些实例会复制 Task 定义的通用行为以及 XYZ 定义的特殊行为。同理，ABC 类的实例也会复制 Task 的行为和 ABC 的行为。在构造完成后，你通常只需要操作这些实例（而不是类），因为每个实例都有你需要完成任务的所有行为。

### 6.1.2 委托理论

但是现在我们试着来使用*委托行为*而不是类来思考同样的问题。

首先你会定义一个名为 Task 的对象（和许多 JavaScript 开发者告诉你的不同，它既不是类也不是函数），它会包含所有任务都可以使用（写作使用，读作委托）的具体行为。接着，对于每个任务（“XYZ”、“ABC”）你都会定义一个对象来存储对应的数据和行为。你会把特定的任务对象都关联到 Task 功能对象上，让它们在需要的时候可以进行委托。

基本上你可以想象成，执行任务“XYZ”需要两个兄弟对象（XYZ 和 Task）协作完成。但是我们并不需要把这些行为放在一起，通过类的复制，我们可以把它们分别放在各自独立的对象中，需要时可以允许 XYZ 对象委托给 Task。

下面是推荐的代码形式，非常简单：

```
Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// 让 XYZ 委托 Task
XYZ = Object.create( Task );

XYZ.prepareTask = function(ID,Label) {
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...
```

在这段代码中，Task 和 XYZ 并不是类（或者函数），它们是对象。XYZ 通过 Object.create(..) 创建，它的 [[Prototype]] 委托了 Task 对象（参见第 5 章）。

相比于面向类（或者说面向对象），我会把这种编码风格称为“对象关联”（OLOO，objects linked to other objects）。我们真正关心的只是 XYZ 对象（和 ABC 对象）委托了 `Task` 对象。

在 JavaScript 中，`[[Prototype]]` 机制会把对象关联到其他对象。无论你多么努力地说服自己，JavaScript 中就是没有类似“类”的抽象机制。这有点像逆流而上：你确实可以这么做，但是如果你选择对抗事实，那要达到目的就显然会更加困难。

对象关联风格的代码还有一些不同之处。

1. 在上面的代码中，`id` 和 `label` 数据成员都是直接存储在 `XYZ` 上（而不是 `Task`）。通常来说，在 `[[Prototype]]` 委托中最好把状态保存在委托者（`XYZ`、`ABC`）而不是委托目标（`Task`）上。

2. 在类设计模式中，我们故意让父类（`Task`）和子类（`XYZ`）中都有 `outputTask` 方法，这样就可以利用重写（多态）的优势。在委托行为中则恰好相反：我们会尽量避免在 `[[Prototype]]` 链的不同级别中使用相同的命名，否则就需要使用笨拙并且脆弱的语法来消除引用歧义（参见第 4 章）。

   这个设计模式要求尽量少使用容易被重写的通用方法名，提倡使用更有描述性的方法名，尤其是要写清相应对象行为的类型。这样做实际上可以创建出更容易理解和维护的代码，因为方法名（不仅在定义的位置，而是贯穿整个代码）更加清晰（自文档）。

3. `this.setID(ID)`；`XYZ` 中的方法首先会寻找 `XYZ` 自身是否有 `setID(..)`，但是 `XYZ` 中并没有这个方法名，因此会通过 `[[Prototype]]` 委托关联到 `Task` 继续寻找，这时就可以找到 `setID(..)` 方法。此外，由于调用位置触发了 `this` 的隐式绑定规则（参见第 2 章），因此虽然 `setID(..)` 方法在 `Task` 中，运行时 `this` 仍然会绑定到 `XYZ`，这正是我们想要的。在之后的代码中我们还会看到 `this.outputID()`，原理相同。

   换句话说，我们和 `XYZ` 进行交互时可以使用 `Task` 中的通用方法，因为 `XYZ` 委托了 `Task`。

委托行为意味着某些对象（`XYZ`）在找不到属性或者方法引用时会把这个请求委托给另一个对象（`Task`）。

这是一种极其强大的设计模式，和父类、子类、继承、多态等概念完全不同。在你的脑海中对象并不是按照父类到子类的关系垂直组织的，而是通过任意方向的委托关联并排组织的。

> 在 API 接口的设计中，委托最好在内部实现，不要直接暴露出去。在之前的例子中我们并没有让开发者通过 API 直接调用 `XYZ.setID()`。（当然，可以这么做！）相反，我们把委托隐藏在了 API 的内部，`XYZ.prepareTask(..)` 会委托 `Task.setID(..)`。更多细节参见 5.4.2 节。

### 1. 互相委托（禁止）

你无法在两个或两个以上互相（双向）委托的对象之间创建*循环委托*。如果你把 B 关联到 A 然后试着把 A 关联到 B，就会出错。

很遗憾（并不是非常出乎意料，但是有点烦人）这种方法是被禁止的。如果你引用了一个两边都不存在的属性或者方法，那就会在 [[Prototype]] 链上产生一个无限递归的循环。但是如果所有的引用都被严格限制的话，B 是可以委托 A 的，反之亦然。因此，互相委托理论上是可以正常工作的，在某些情况下这是非常有用的。

之所以要禁止互相委托，是因为引擎的开发者们发现在设置时检查（并禁止！）一次无限循环引用要更加高效，否则每次从对象中查找属性时都需要进行检查。

### 2. 调试

我们来简单介绍一个容易让开发者感到迷惑的细节。通常来说，JavaScript 规范并不会控制浏览器中开发者工具对于特定值或者结构的表示方式，浏览器和引擎可以自己选择合适的方式来进行解析，因此浏览器和工具的解析结果*并不一定相同*。比如，下面这段代码的结果只能在 Chrome 的开发者工具中才能看到。

这段传统的"类构造函数"JavaScript 代码在 Chrome 开发者工具的*控制台*中结果如下所示：

```
function Foo() {}

var a1 = new Foo();

a1; // Foo {}
```

我们看代码的最后一行：表达式 a1 的输出是 Foo {}。如果你在 Firefox 中运行同样的代码会得到 Object {}。为什么会这样呢？这些输出是什么意思呢？

Chrome 实际上想说的是"{} 是一个空对象，由名为 Foo 的函数构造"。Firefox 想说的是"{} 是一个空对象，由 Object 构造"。之所以有这种细微的差别，是因为 Chrome 会动态跟踪并把实际执行构造过程的函数名当作一个*内置属性*，但是其他浏览器并不会跟踪这些额外的信息。

看起来可以用 JavaScript 的机制来解释 Chrome 的跟踪原理：

```
function Foo() {}

var a1 = new Foo();

a1.constructor; // Foo(){}
a1.constructor.name; // "Foo"
```

Chrome 是不是直接输出了对象的 .constructor.name 呢？令人迷惑的是，答案是"既是又不是"。

思考下面的代码：

```
function Foo() {}

var a1 = new Foo();

Foo.prototype.constructor = function Gotcha(){};

a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"

a1; // Foo {}
```

即使我们把 `a1.constructor.name` 修改为另一个合理的值（`Gotcha`），Chrome 控制台仍然会输出 `Foo`。

看起来之前那个问题（是否使用 `.constructor.name`？）的答案是“不是”；Chrome 在内部肯定是通过另一种方式进行跟踪。

别着急！我们先看看下面这段代码：

```
var Foo = {};

var a1 = Object.create( Foo );

a1; // Object {}

Object.defineProperty( Foo, "constructor", {
    enumerable: false,
    value: function Gotcha(){}
});

a1; // Gotcha {}
```

啊哈！抓到你了（Gotcha 的意思就是抓到你了）！本例中 Chrome 的控制台*确实*使用了 `.constructor.name`。实际上，在编写本书时，这个行为被认定是 Chrome 的一个 bug，当你读到此书时，它可能已经被修复了。所以你看到的可能是 `a1; // Object {}`。

除了这个 bug，Chrome 内部跟踪（只用于调试输出）“构造函数名称”的方法是 Chrome 自身的一种扩展行为，并不包含在 JavaScript 的规范中。

如果你并不是使用“构造函数”来生成对象，比如使用本章介绍的对象关联风格来编写代码，那 Chrome 就*无法*跟踪对象内部的“构造函数名称”，这样的对象输出是 `Object {}`，意思是“`Object()` 构造出的对象”。

当然，这并不是对象关联风格代码的缺点。当你使用对象关联风格来编写代码并使用行为委托设计模式时，并不需要关注是谁“构造了”对象（就是使用 `new` 调用的那个*函数*）。只有使用类风格来编写代码时 Chrome 内部的“构造函数名称”跟踪才有意义，使用对象关联时这个功能不起任何作用。

### 6.1.3　比较思维模型

现在你已经明白了“类”和“委托”这两种设计模式的理论区别，接下来我们看看它们在思维模型方面的区别。

我们会通过一些示例（Foo、Bar）代码来比较一下两种设计模式（面向对象和对象关联）具体的实现方法。下面是典型的（“原型”）面向对象风格：

```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call( this, who );
}
Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

子类 Bar 继承了父类 Foo，然后生成了 b1 和 b2 两个实例。b1 委托了 Bar.prototype，后者委托了 Foo.prototype。这种风格很常见，你应该很熟悉了。

下面我们看看如何使用对象关联风格来编写功能完全相同的代码：

```
Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};
Bar = Object.create( Foo );

Bar.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
```

```
b2.init( "b2" );

b1.speak();
b2.speak();
```

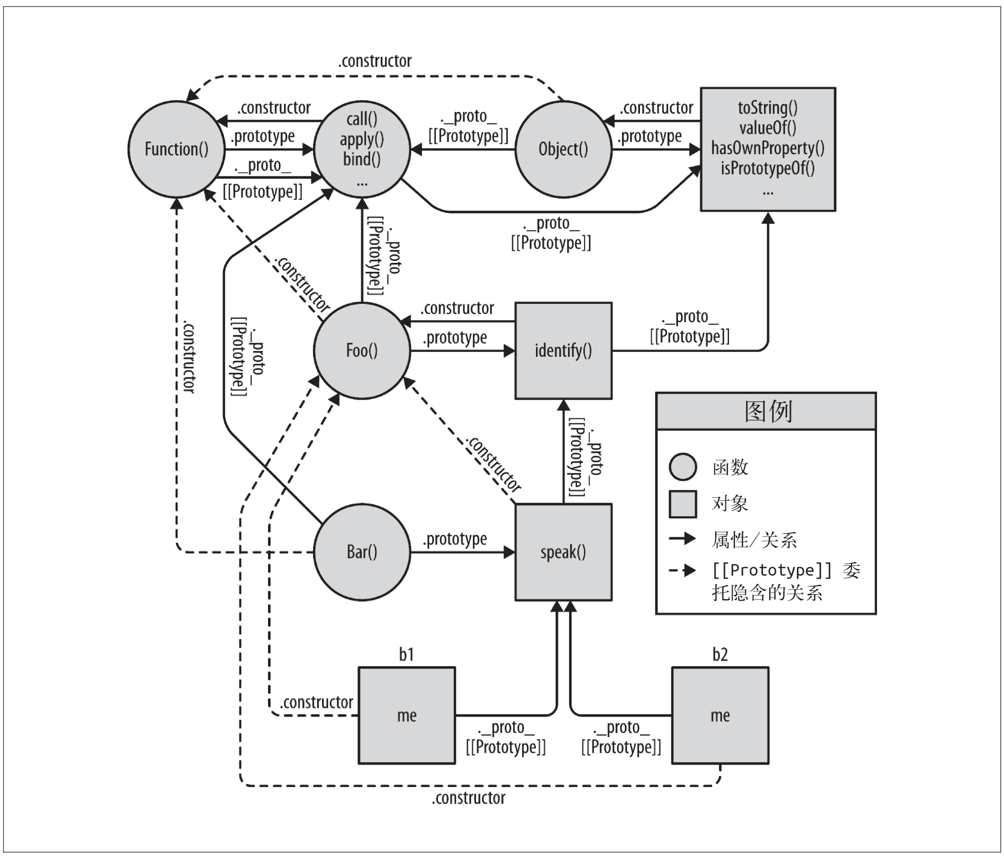
这段代码中我们同样利用 [[Prototype]] 把 b1 委托给 Bar 并把 Bar 委托给 Foo，和上一段代码一模一样。我们仍然实现了三个对象之间的关联。

但是非常重要的一点是，这段代码简洁了许多，我们只是把对象关联起来，并不需要那些既复杂又令人困惑的模仿类的行为（构造函数、原型以及 new）。

问问你自己：如果对象关联风格的代码能够实现类风格代码的所有功能并且更加简洁易懂，那它是不是比类风格更好？

下面我们看看两段代码对应的思维模型。

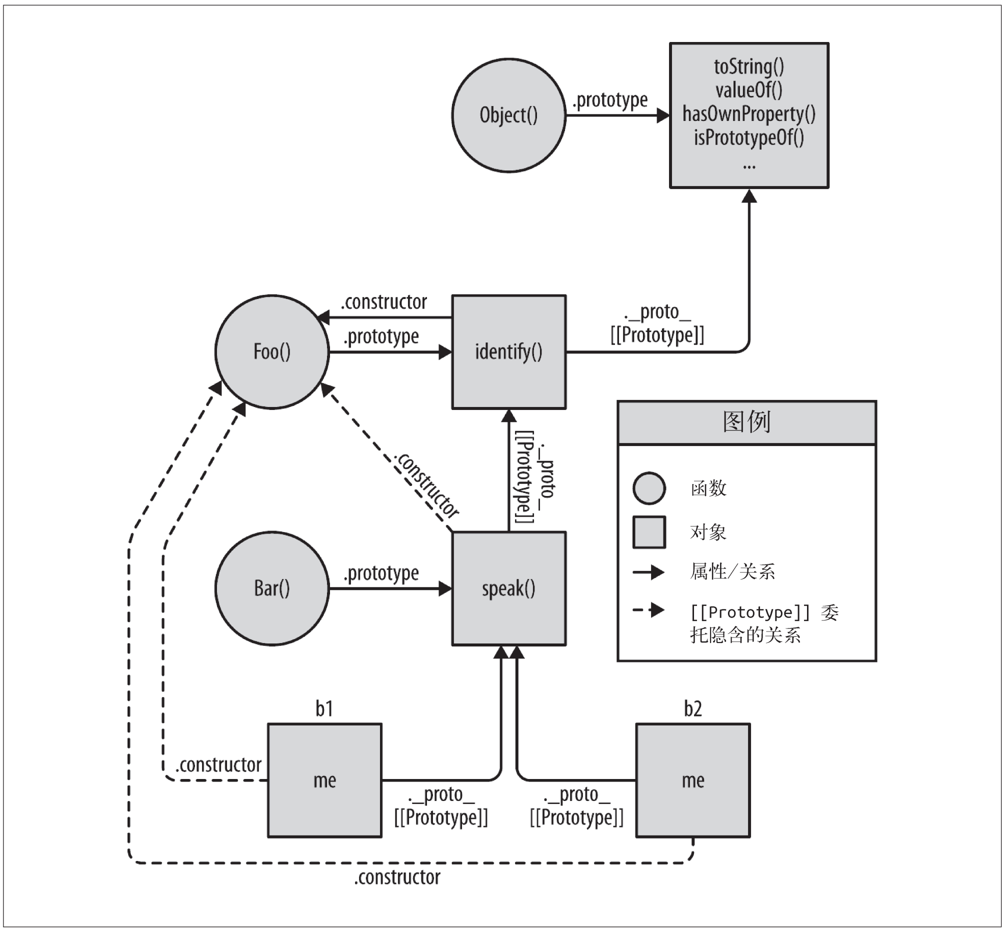
首先，类风格代码的思维模型强调实体以及实体间的关系：



实际上这张图有点不清晰 / 误导人，因为它还展示了许多技术角度不需要关注的细节（但是你必须理解它们）！从图中可以看出这是一张十分复杂的关系网。此外，如果你跟着图中的箭头走就会发现，JavaScript 机制有很强的内部连贯性。

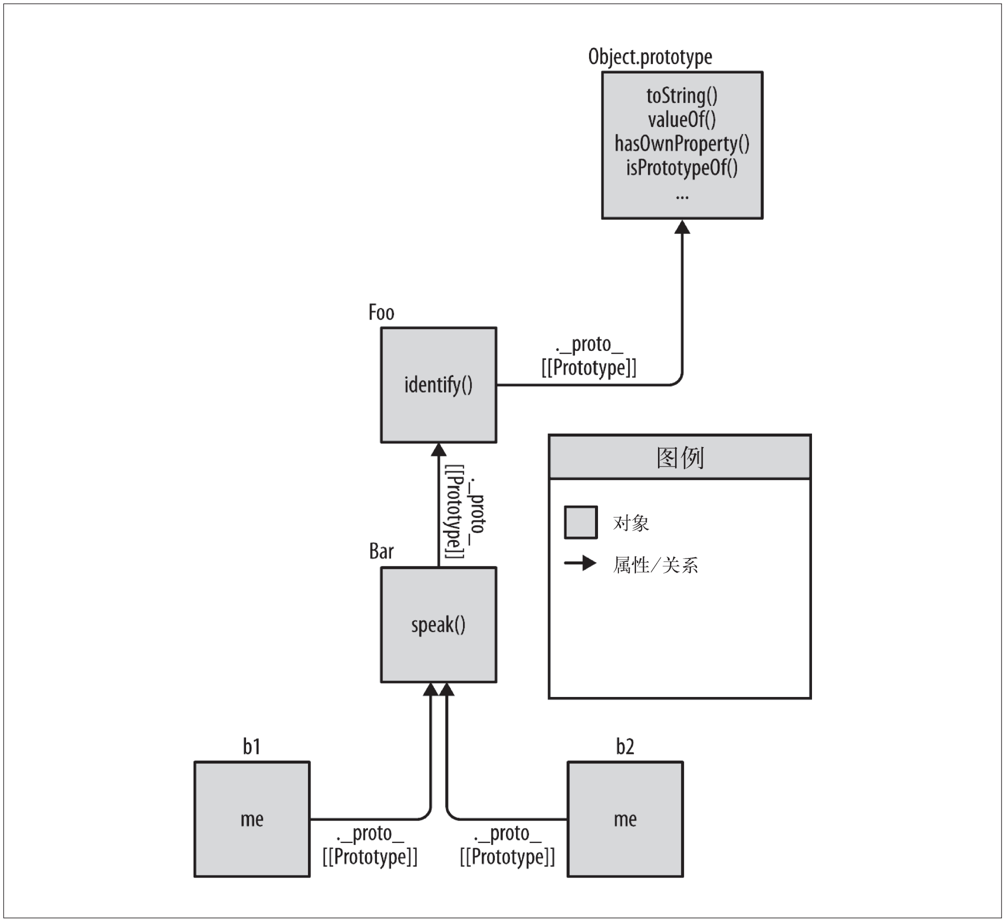
举例来说，JavaScript 中的函数之所以可以访问 `call(..)`、`apply(..)` 和 `bind(..)`（参见第 2 章），就是因为函数本身是对象。而函数对象同样有 `[[Prototype]]` 属性并且关联到 `Function.prototype` 对象，因此所有函数对象都可以通过委托调用这些默认方法。JavaScript 能做到这一点，你也可以！

好，下面我们来看一张简化版的图，它更“清晰”一些——只展示了必要的对象和关系：




仍然很复杂，是吧？虚线表示的是 `Bar.prototype` 继承 `Foo.prototype` 之后丢失的 `.constructor` 属性引用（参见 5.2.3 节的“回顾‘构造函数’”部分），它们还没有被修复。即使移除这些虚线，这个思维模型在你处理对象关联时仍然非常复杂。

现在我们看看对象关联风格代码的思维模型：




通过比较可以看出，对象关联风格的代码显然更加简洁，因为这种代码只关注一件事：对象之间的关联关系。

其他的“类”技巧都是非常复杂并且令人困惑的。去掉它们之后，事情会变得简单许多（同时保留所有功能）。

## 6.2 类与对象

我们已经看到了“类”和“行为委托”在理论和思维模型方面的区别，现在看看在真实场景中如何应用这些方法。

首先看看 Web 开发中非常典型的一种前端场景：创建 UI 控件（按钮、下拉列表，等等）。

## 6.2.1 控件“类”

你可能已经习惯了面向对象设计模式，所以很快会想到一个包含所有通用控件行为的父类（可能叫作 `Widget`）和继承父类的特殊控件子类（比如 `Button`）。

> 这里将使用 jQuery 来操作 DOM 和 CSS，因为这些操作和我们现在讨论的内容没有关系。这些代码并不关注你是否使用，或使用哪种 JavaScript 框架（jQuery、Dojo、YUI，等等）来解决问题。

下面这段代码展示的是如何在不使用任何“类”辅助库或者语法的情况下，使用纯 JavaScript 实现类风格的代码：

```
// 父类
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        } ).appendTo( $where );
    }
};

// 子类
function Button(width,height,label) {
    // 调用“super”构造函数
    Widget.call( this, width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
}

// 让 Button“继承”Widget
Button.prototype = Object.create( Widget.prototype );

// 重写 render(..)
Button.prototype.render = function($where) {
    //“super”调用
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
```

```
};

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

在面向对象设计模式中我们需要先在父类中定义基础的 `render(..)`，然后在子类中重写它。子类并不会替换基础的 `render(..)`，只是添加一些按钮特有的行为。

可以看到代码中出现了丑陋的显式伪多态（参见第 4 章），即通过 `Widget.call` 和 `Widget.prototype.render.call` 从“子类”方法中引用“父类”中的基础方法。呸！

### ES6的class语法糖

附录 A 会详细介绍 ES6 的 `class` 语法糖，不过这里可以简单介绍一下如何使用 `class` 来实现相同的功能：

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}
```

```
$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

毫无疑问，使用 ES6 的 `class` 之后，上一段代码中许多丑陋的语法都不见了，`super(..)` 函数棒极了。（尽管深入探究就会发现并不是那么完美！）

尽管语法上得到了改进，但实际上这里并没有真正的类，`class` 仍然是通过 `[[Prototype]]` 机制实现的，因此我们仍然面临第 4 章至第 6 章提到的思维模式不匹配问题。附录 A 会详细介绍 ES6 的 `class` 语法及其实现细节，我们会看到为什么解决语法上的问题无法真正解除对于 JavaScript 中类的误解，尽管它看起来非常像一种解决办法！

无论你使用的是传统的原型语法还是 ES6 中的新语法糖，你仍然需要用“类”的概念来对问题（UI 控件）进行建模。就像前几章试图证明的一样，这种做法会为你带来新的麻烦。

## 6.2.2 委托控件对象

下面的例子使用对象关联风格委托来更简单地实现 `Widget/Button`：

```
var Widget = {
    init: function(width,height){
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    },
    insert: function($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
};

var Button = Object.create( Widget );

Button.setup = function(width,height,label){
    // 委托调用
    this.init( width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
};
Button.build = function($where) {
```

```
        // 委托调用
        this.insert( $where );
        this.$elem.click( this.onClick.bind( this ) );
    };
    Button.onClick = function(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    };

    $( document ).ready( function(){
        var $body = $( document.body );

        var btn1 = Object.create( Button );
        btn1.setup( 125, 30, "Hello" );

        var btn2 = Object.create( Button );
        btn2.setup( 150, 40, "World" );

        btn1.build( $body );
        btn2.build( $body );
    } );
```

使用对象关联风格来编写代码时不需要把 Widget 和 Button 当作父类和子类。相反，Widget 只是一个对象，包含一组通用的函数，任何类型的控件都可以委托，Button 同样只是一个对。（当然，它会通过委托关联到 Widget ！）

从设计模式的角度来说，我们并没有像类一样在两个对象中都定义相同的方法名 render(..)，相反，我们定义了两个更具描述性的方法名（insert(..) 和 build(..)）。同理，*初始化*方法分别叫作 init(..) 和 setup(..)。

在委托设计模式中，除了建议使用不相同并且更具描述性的方法名之外，还要通过对象关联避免丑陋的显式伪多态调用（Widget.call 和 Widget.prototype.render.call），代之以简单的相对委托调用 this.init(..) 和 this.insert(..)。

从语法角度来说，我们同样没有使用任何构造函数、.prototype 或 new，实际上也没必要使用它们。

如果你仔细观察就会发现，之前的一次调用（var btn1 = new Button(..)）现在变成了两次（var btn1 = Object.create(Button) 和 btn1.setup(..)）。乍一看这似乎是一个缺点（需要更多代码）。

但是这一点其实也是对象关联风格代码相比传统原型风格代码有优势的地方。为什么呢？

使用类构造函数的话，你需要（并不是硬性要求，但是强烈建议）在同一个步骤中实现构造和初始化。然而，在许多情况下把这两步分开（就像对象关联代码一样）更灵活。

举例来说，假如你在程序启动时创建了一个实例池，然后一直等到实例被取出并使用时才执行特定的初始化过程。这个过程中两个函数调用是挨着的，但是完全可以根据需要让它

们出现在不同的位置。

对象关联可以更好地支持关注分离（separation of concerns）原则，创建和初始化并不需要合并为一个步骤。

## 6.3　更简洁的设计

对象关联除了能让代码看起来更简洁（并且更具扩展性）外还可以通过行为委托模式简化代码结构。我们来看最后一个例子，它展示了对象关联如何简化整体设计。

在这个场景中我们有两个控制器对象，一个用来操作网页中的登录表单，另一个用来与服务器进行验证（通信）。

我们需要一个辅助函数来创建 Ajax 通信。我们使用的是 jQuery（尽管其他框架也做得不错），它不仅可以处理 Ajax 并且会返回一个类 Promise 的结果，因此我们可以使用 `.then(..)` 来监听响应。

> 这里我们不会介绍 Promise，但是在本系列之后的书中会介绍。

在传统的类设计模式中，我们会把基础的函数定义在名为 `Controller` 的类中，然后派生两个子类 `LoginController` 和 `AuthController`，它们都继承自 `Controller` 并且重写了一些基础行为：

```
// 父类
function Controller() {
    this.errors = [];
}
Controller.prototype.showDialog(title,msg) {
    // 给用户显示标题和消息
};
Controller.prototype.success = function(msg) {
    this.showDialog( "Success", msg );
};
Controller.prototype.failure = function(err) {
    this.errors.push( err );
    this.showDialog( "Error", err );
};

// 子类
function LoginController() {
    Controller.call( this );
}
// 把子类关联到父类
```

```
LoginController.prototype =
    Object.create( Controller.prototype );
LoginController.prototype.getUser = function() {
    return document.getElementById( "login_username" ).value;
};
LoginController.prototype.getPassword = function() {
    return document.getElementById( "login_password" ).value;
};
LoginController.prototype.validateEntry = function(user,pw) {
    user = user || this.getUser();
    pw = pw || this.getPassword();

    if (!(user && pw)) {
        return this.failure(
           "Please enter a username & password!"
        );
    }
    else if (user.length < 5) {
        return this.failure(
           "Password must be 5+ characters!"
        );
    }

    // 如果执行到这里说明通过验证
    return true;
};
// 重写基础的 failure()
LoginController.prototype.failure = function(err) {
    //“super”调用
    Controller.prototype.failure.call(
       this,
       "Login invalid: " + err
    );
};

// 子类
function AuthController(login) {
    Controller.call( this );
    // 合成
    this.login = login;
}
// 把子类关联到父类
AuthController.prototype =
   Object.create( Controller.prototype );
AuthController.prototype.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.prototype.checkAuth = function() {
    var user = this.login.getUser();
    var pw = this.login.getPassword();

    if (this.login.validateEntry( user, pw )) {
```

```
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.success.bind( this ) )
        .fail( this.failure.bind( this ) );
    }
};
// 重写基础的 success()
AuthController.prototype.success = function() {
    //“super”调用
    Controller.prototype.success.call( this, "Authenticated!" );
};
// 重写基础的 failure()
AuthController.prototype.failure = function(err) {
    //“super”调用
    Controller.prototype.failure.call(
        this,
        "Auth Failed: " + err
    );
};

var auth = new AuthController();
auth.checkAuth(
    // 除了继承，我们还需要合成
    new LoginController()
);
```

所有控制器共享的基础行为是 success(..)、failure(..) 和 showDialog(..)。子类 LoginController 和 AuthController 通过重写 failure(..) 和 success(..) 来扩展默认基础类行为。此外，注意 AuthController 需要一个 LoginController 的实例来和登录表单进行交互，因此这个实例变成了一个数据属性。

另一个需要注意的是我们在继承的基础上进行了一些合成。AuthController 需要使用 LoginController，因此我们实例化后者（new LoginController()）并用一个类成员属性 this.login 来引用它，这样 AuthController 就可以调用 LoginController 的行为。

> 你可能想让 AuthController 继承 LoginController 或者相反，这样我们就通过继承链实现了真正的合成。但是这就是类继承在问题领域建模时会产生的问题，因为 AuthController 和 LoginController 都不具备对方的基础行为，所以这种继承关系是不恰当的。我们的解决办法是进行一些简单的合成从而让它们既不必互相继承又可以互相合作。

如果你熟悉面向类设计，你一定会觉得以上内容非常亲切和自然。

## 反类

但是，我们真的需要用一个 Controller 父类、两个子类加上合成来对这个问题进行建模吗？能不能使用对象关联风格的行为委托来实现更简单的设计呢？当然可以！

```
var LoginController = {
    errors: [],
    getUser: function() {
        return document.getElementById(
           "login_username"
        ).value;
    },
    getPassword: function() {
        return document.getElementById(
           "login_password"
        ).value;
    },
    validateEntry: function(user,pw) {
        user = user || this.getUser();
        pw = pw || this.getPassword();

        if (!(user && pw)) {
            return this.failure(
               "Please enter a username & password!"
            );
        }
        else if (user.length < 5) {
            return this.failure(
               "Password must be 5+ characters!"
            );
        }

        // 如果执行到这里说明通过验证
        return true;
    },
    showDialog: function(title,msg) {
        // 给用户显示标题和消息
    },
    failure: function(err) {
        this.errors.push( err );
        this.showDialog( "Error", "Login invalid: " + err );
    }
};
// 让 AuthController 委托 LoginController
var AuthController = Object.create( LoginController );

AuthController.errors = [];
AuthController.checkAuth = function() {
    var user = this.getUser();
    var pw = this.getPassword();

    if (this.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
```

```
            pw: pw
        } )
        .then( this.accepted.bind( this ) )
        .fail( this.rejected.bind( this ) );
    }
};
AuthController.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.accepted = function() {
    this.showDialog( "Success", "Authenticated!" )
};
AuthController.rejected = function(err) {
    this.failure( "Auth Failed: " + err );
};
```

由于 AuthController 只是一个对象（LoginController 也一样），因此我们不需要实例化（比如 new AuthController()），只需要一行代码就行：

```
AuthController.checkAuth();
```

借助对象关联，你可以简单地向委托链上添加一个或多个对象，而且同样不需要实例化：

```
var controller1 = Object.create( AuthController );
var controller2 = Object.create( AuthController );
```

在行为委托模式中，AuthController 和 LoginController 只是对象，它们之间是*兄弟关系*，并不是父类和子类的关系。代码中 AuthController 委托了 LoginController，反向委托也完全没问题。

这种模式的重点在于只需要两个实体（LoginController 和 AuthController），而之前的模式需要三个。

我们不需要 Controller 基类来"共享"两个实体之间的行为，因为委托足以满足我们需要的功能。同样，前面提到过，我们也不需要实例化类，因为它们根本就不是类，它们只是对象。此外，我们也不需要合成，因为两个对象可以通过委托进行合作。

最后，我们避免了面向类设计模式中的多态。我们在不同的对象中没有使用相同的函数名 success(..) 和 failure(..)，这样就不需要使用丑陋的显示伪多态。相反，在 AuthController 中它们的名字是 accepted(..) 和 rejected(..)——可以更好地描述它们的行为。

总结：我们用一种（极其）简单的设计实现了同样的功能，这就是对象关联风格代码和*行为委托*设计模式的力量。

## 6.4 更好的语法

ES6 的 class 语法可以简洁地定义类方法，这个特性让 class 乍看起来更有吸引力（附录 A 会介绍为什么要避免使用这个特性）：

```
class Foo {
    methodName() { /* .. */ }
}
```

我们终于可以抛弃定义中的关键字 function 了，对所有 JavaScript 开发者来说真是大快人心！

你可能注意到了，在之前推荐的对象关联语法中出现了许多 function，看起来违背了对象关联的简洁性。但是实际上大可不必如此！

在 ES6 中我们可以在任意对象的字面形式中使用*简洁方法声明*（concise method declaration），所以对象关联风格的对象可以这样声明（和 class 的语法糖一样）：

```
var LoginController = {
    errors: [],
    getUser() { // 妈妈再也不用担心代码里有 function 了！
        // ...
    },
    getPassword() {
        // ...
    }
    // ...
};
```

唯一的区别是对象的字面形式仍然需要使用“,”来分隔元素，而 class 语法不需要。这个区别对于整体的设计来说无关紧要。

此外，在 ES6 中，你可以使用对象的字面形式（这样就可以使用简洁方法定义）来改写之前繁琐的属性赋值语法（比如 AuthController 的定义），然后用 Object.setPrototypeOf(..) 来修改它的 [[Prototype]]：

```
// 使用更好的对象字面形式语法和简洁方法
var AuthController = {
    errors: [],
    checkAuth() {
        // ...
    },
    server(url,data) {
        // ...
    }
    // ...
};

// 现在把 AuthController 关联到 LoginController
Object.setPrototypeOf( AuthController, LoginController );
```

使用 ES6 的简洁方法可以让对象关联风格更加人性化（并且仍然比典型的原型风格代码更加简洁和优秀）。你完全不需要使用类就能享受整洁的对象语法！

## 反词法

简洁方法有一个非常小但是非常重要的缺点。思考下面的代码：

```
var Foo = {
    bar() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

去掉语法糖之后的代码如下所示：

```
var Foo = {
    bar: function() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

看到区别了吗？由于函数对象本身没有名称标识符，所以 `bar()` 的缩写形式（`function()..`）实际上会变成一个*匿名函数表达式*并赋值给 `bar` 属性。相比之下，*具名函数表达式*（`function baz()..`）会额外给 `.baz` 属性附加一个词法名称标识符 `baz`。

然后呢？在本书第一部分“作用域和闭包”中我们分析了匿名函数表达式的三大主要缺点，下面我们会简单介绍一下这三个缺点，然后和简洁方法定义进行对比。

匿名函数没有 `name` 标识符，这会导致：

1. 调试栈更难追踪；
2. 自我引用（递归、事件（解除）绑定，等等）更难；
3. 代码（稍微）更难理解。

简洁方法没有第 1 和第 3 个缺点。

去掉语法糖的版本使用的是*匿名函数表达式*，通常来说并不会在追踪栈中添加 `name`，但是简洁方法很特殊，会给对应的函数对象设置一个内部的 `name` 属性，这样理论上可以用在追踪栈中。（但是追踪的具体实现是不同的，因此无法保证可以使用。）

很不幸，简洁方法无法避免第 2 个缺点，它们不具备可以自我引用的词法标识符。思考下面的代码：

```
var Foo = {
    bar: function(x) {
        if(x<10){
           return Foo.bar( x * 2 );
        }
```

```
            return x;
        },
        baz: function baz(x) {
            if(x < 10){
                return baz( x * 2 );
            }
            return x;
        }
    };
```

在本例中使用 Foo.bar(x*2) 就足够了，但是在许多情况下无法使用这种方法，比如多个对象通过代理共享函数、使用 this 绑定，等等。这种情况下最好的办法就是使用函数对象的 name 标识符来进行真正的自我引用。

使用简洁方法时一定要小心这一点。如果你需要自我引用的话，那最好使用传统的具名函数表达式来定义对应的函数（· baz: function baz(){..}·），不要使用简洁方法。

## 6.5 内省

如果你写过许多面向类的程序（无论是使用 JavaScript 还是其他语言），那你可能很熟悉自省。自省就是检查实例的类型。类实例的自省主要目的是通过创建方式来判断对象的结构和功能。

下面的代码使用 instanceof（参见第 5 章）来推测对象 a1 的功能：

```
function Foo() {
    // ...
}
Foo.prototype.something = function(){
    // ...
}

var a1 = new Foo();

// 之后

if (a1 instanceof Foo) {
    a1.something();
}
```

因为 Foo.prototype（不是 Foo！）在 a1 的 [[Prototype]] 链上（参见第 5 章），所以 instanceof 操作（会令人困惑地）告诉我们 a1 是 Foo“类”的一个实例。知道了这点后，我们就可以认为 a1 有 Foo“类”描述的功能。

当然，Foo 类并不存在，只有一个普通的函数 Foo，它引用了 a1 委托的对象（Foo.prototype）。从语法角度来说，instanceof 似乎是检查 a1 和 Foo 的关系，但是实际上它想说的是 a1 和 Foo.prototype（引用的对象）是互相关联的。

instanceof 语法会产生语义困惑而且非常不直观。如果你想检查对象 a1 和某个对象的关系，那必须使用另一个引用该对象的函数才行——你不能直接判断两个对象是否关联。

还记得本章之前介绍的抽象的 Foo/Bar/b1 例子吗，简单来说是这样的：

```
function Foo() { /* .. */ }
Foo.prototype...

function Bar() { /* .. */ }
Bar.prototype = Object.create( Foo.prototype );

var b1 = new Bar( "b1" );
```

如果要使用 instanceof 和 .prototype 语义来检查本例中实体的关系，那必须这样做：

```
// 让 Foo 和 Bar 互相关联
Bar.prototype instanceof Foo; // true
Object.getPrototypeOf( Bar.prototype )
   === Foo.prototype; // true
Foo.prototype.isPrototypeOf( Bar.prototype ); // true

// 让 b1 关联到 Foo 和 Bar
b1 instanceof Foo; // true
b1 instanceof Bar; // true
Object.getPrototypeOf( b1 ) === Bar.prototype; // true
Foo.prototype.isPrototypeOf( b1 ); // true
Bar.prototype.isPrototypeOf( b1 ); // true
```

显然这是一种非常糟糕的方法。举例来说，（使用类时）你最直观的想法可能是使用 Bar instanceof Foo（因为很容易把“实例”理解成“继承”），但是在 JavaScript 中这是行不通的，你必须使用 Bar.prototype instanceof Foo。

还有一种常见但是可能更加脆弱的内省模式，许多开发者认为它比 instanceof 更好。这种模式被称为“鸭子类型”。这个术语源自这句格言“如果看起来像鸭子，叫起来像鸭子，那就一定是鸭子。”

举例来说：

```
if (a1.something) {
    a1.something();
}
```

我们并没有检查 a1 和委托 something() 函数的对象之间的关系，而是假设如果 a1 通过了测试 a1.something 的话，那 a1 就一定能调用 .something()（无论这个方法存在于 a1 自身还是委托到其他对象）。这个假设的风险其实并不算很高。

但是“鸭子类型”通常会在测试之外做出许多关于对象功能的假设，这当然会带来许多风险（或者说脆弱的设计）。

ES6 的 Promise 就是典型的“鸭子类型”（之前解释过，本书并不会介绍 Promise）。

出于各种各样的原因，我们需要判断一个对象引用是否是 Promise，但是判断的方法是检查对象是否有 then() 方法。换句话说，如果对象有 then() 方法，ES6 的 Promise 就会认为这个对象是“可持续”（thenable）的，因此会期望它具有 Promise 的所有标准行为。

如果有一个不是 Promise 但是具有 then() 方法的对象，那你千万不要把它用在 ES6 的 Promise 机制中，否则会出错。

这个例子清楚地解释了“鸭子类型”的危害。你应该尽量避免使用这个方法，即使使用也要保证条件是可控的。

现在回到本章想说的对象关联风格代码，其内省更加简洁。我们先来回顾一下之前的 Foo/Bar/b1 对象关联例子（只包含关键代码）：

```
var Foo = { /* .. */ };

var Bar = Object.create( Foo );
Bar...

var b1 = Object.create( Bar );
```

使用对象关联时，所有的对象都是通过 [[Prototype]] 委托互相关联，下面是内省的方法，非常简单：

```
// 让 Foo 和 Bar 互相关联
Foo.isPrototypeOf( Bar ); // true
Object.getPrototypeOf( Bar ) === Foo; // true

// 让 b1 关联到 Foo 和 Bar
Foo.isPrototypeOf( b1 ); // true
Bar.isPrototypeOf( b1 ); // true
Object.getPrototypeOf( b1 ) === Bar; // true
```

我们没有使用 instanceof，因为它会产生一些和类有关的误解。现在我们想问的问题是“你是我的原型吗？”我们并不需要使用间接的形式，比如 Foo.prototype 或者繁琐的 Foo.prototype.isPrototypeOf(..)。

我觉得和之前的方法比起来，这种方法显然更加简洁并且清晰。再说一次，我们认为 JavaScript 中对象关联比类风格的代码更加简洁（而且功能相同）。

## 6.6　小结

在软件架构中你可以选择是否使用类和继承设计模式。大多数开发者理所当然地认为类是唯一（合适）的代码组织方式，但是本章中我们看到了另一种更少见但是更强大的设计模

式：行为委托。

行为委托认为对象之间是兄弟关系，互相委托，而不是父类和子类的关系。JavaScript 的 [[Prototype]] 机制本质上就是行为委托机制。也就是说，我们可以选择在 JavaScript 中努力实现类机制（参见第 4 和第 5 章），也可以拥抱更自然的 [[Prototype]] 委托机制。

当你只用对象来设计代码时，不仅可以让语法更加简洁，而且可以让代码结构更加清晰。

对象关联（对象之前互相关联）是一种编码风格，它倡导的是直接创建和关联对象，不把它们抽象成类。对象关联可以用基于 [[Prototype]] 的行为委托非常自然地实现。

# 附录 A
# ES6中的Class

可以用一句话总结本书的第二部分（第 4 章至第 6 章）：类是一种可选（而不是必须）的设计模式，而且在 JavaScript 这样的 `[[Prototype]]` 语言中实现类是很别扭的。

这种别扭的感觉不只是来源于语法，虽然语法是很重要的原因。第 4 章和第 5 章介绍了许多语法的缺点：繁琐杂乱的 `.prototype` 引用、试图调用原型链上层同名函数时的显式伪多态（参见第 4 章）以及不可靠、不美观而且容易被误解成"构造函数"的 `.constructor`。

除此之外，类设计其实还存在更深刻的问题。第 4 章指出，传统面向类的语言中父类和子类、子类和实例之间其实是复制操作，但是在 `[[Prototype]]` 中并没有复制，相反，它们之间只有委托关联。

对象关联代码和行为委托（参见第 6 章）使用了 `[[Prototype]]` 而不是将它藏起来，对比其简洁性可以看出，类并不适用于 JavaScript。

## A.1 class

不过我们并不需要再纠结于这个问题，这里提到只是让你简单回忆一下；现在我们来看看 ES6 的 `class` 机制。我们会介绍它的工作原理并分析 `class` 是否改进了之前提到的那些缺点。

首先回顾一下第 6 章中的 `Widget/Button` 例子：

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
```

```
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}
```

除了语法更好看之外，ES6 还解决了什么问题呢？

1. （基本上，下面会详细介绍）不再引用杂乱的 .prototype 了。
2. Button 声明时直接“继承”了 Widget，不再需要通过 Object.create(..) 来替换 .prototype 对象，也不需要设置 .__proto__ 或者 Object.setPrototypeOf(..)。
3. 可以通过 super(..) 来实现相对多态，这样任何方法都可以引用原型链上层的同名方法。这可以解决第 4 章提到过的那个问题：构造函数不属于类，所以无法互相引用——super() 可以完美解决构造函数的问题。
4. class 字面语法不能声明属性（只能声明方法）。看起来这是一种限制，但是它会排除掉许多不好的情况，如果没有这种限制的话，原型链末端的“实例”可能会意外地获取其他地方的属性（这些属性隐式被所有“实例”所“共享”）。所以，class 语法实际上可以帮助你避免犯错。
5. 可以通过 extends 很自然地扩展对象（子）类型，甚至是内置的对象（子）类型，比如 Array 或 RegExp。没有 class ..extends 语法时，想实现这一点是非常困难的，基本上只有框架的作者才能搞清楚这一点。但是现在可以轻而易举地做到！

平心而论，class 语法确实解决了典型原型风格代码中许多显而易见的（语法）问题和缺点。

## A.2 class陷阱

然而，class 语法并没有解决所有的问题，在 JavaScript 中使用“类”设计模式仍然存在许多深层问题。

首先，你可能会认为 ES6 的 class 语法是向 JavaScript 中引入了一种新的“类”机制，其实不是这样。class 基本上只是现有 [[Prototype]]（委托！）机制的一种语法糖。

也就是说，class 并不会像传统面向类的语言一样在声明时静态复制所有行为。如果你（有意或无意）修改或者替换了父“类”中的一个方法，那子“类”和所有实例都会受到影响，因为它们在定义时并没有进行复制，只是使用基于 [[Prototype]] 的实时委托：

```
class C {
    constructor() {
        this.num = Math.random();
    }
    rand() {
        console.log( "Random: " + this.num );
    }
}

var c1 = new C();
c1.rand(); // "Random: 0.4324299..."

C.prototype.rand = function() {
    console.log( "Random: " + Math.round( this.num * 1000 ));
};

var c2 = new C();
c2.rand(); // "Random: 867"

c1.rand(); // "Random: 432" ——噢!
```

如果你已经明白委托的原理所以并不会期望得到“类”的副本的话，那这种行为才看起来比较合理。所以你需要问自己：为什么要使用本质上不是类的 class 语法呢？

ES6 中的 class 语法不是会让传统类和委托对象之间的区别更加难以发现和理解吗？

class 语法无法定义类成员属性（只能定义方法），如果为了跟踪实例之间共享状态必须要这么做，那你只能使用丑陋的 .prototype 语法，像这样：

```
class C {
    constructor() {
        // 确保修改的是共享状态而不是在实例上创建一个屏蔽属性!
        C.prototype.count++;

        // this.count 可以通过委托实现我们想要的功能
        console.log( "Hello: " + this.count );
    }
```

```
}

// 直接向 prototype 对象上添加一个共享状态
C.prototype.count = 0;

var c1 = new C();
// Hello: 1

var c2 = new C();
// Hello: 2

c1.count === 2; // true
c1.count === c2.count; // true
```

这种方法最大的问题是，它违背了 class 语法的本意，在实现中暴露（泄露！）了 .prototype。

如果使用 this.count++ 的话，我们会很惊讶地发现在对象 c1 和 c2 上都创建了 .count 属性，而不是更新共享状态。class 没有办法解决这个问题，并且干脆就不提供相应的语法支持，所以你根本就不应该这样做。

此外，class 语法仍然面临意外屏蔽的问题：

```
class C {
    constructor(id) {
        // 噢，郁闷，我们的 id 属性屏蔽了 id() 方法
        this.id = id;
    }
    id() {
        console.log( "Id: " + id );
    }
}

var c1 = new C( "c1" );
c1.id(); // TypeError -- c1.id 现在是字符串 "c1"
```

除此之外，super 也存在一些非常细微的问题。你可能认为 super 的绑定方法和 this 类似（参见第 2 章），也就是说，无论目前的方法在原型链中处于什么位置，super 总会绑定到链中的上一层。

然而，出于性能考虑（this 绑定已经是很大的开销了），super 并不是动态绑定的，它会在声明时"静态"绑定。没什么大不了的，是吧？

呃……可能，可能不是这样。如果你和大多数 JavaScript 开发者一样，会用许多不同的方法把函数应用在不同的（使用 class 定义的）对象上，那你可能不知道，每次执行这些操作时都必须重新绑定 super。

此外，根据应用方式的不同，super 可能不会绑定到合适的对象（至少和你想的不一样），

所以你可能（写作本书时，TC39 正在讨论这个话题）需要用 toMethod(..) 来手动绑定 super（类似用 bind(..) 来绑定 this——参见第 2 章）。

你已经习惯了把方法应用到不同的对象上，从而可以自动利用 this 的隐式绑定规则（参见第 2 章）。但是这对于 super 来说是行不通的。

思考下面代码中 super 的行为（D 和 E 上）：

```
class P {
    foo() { console.log( "P.foo" ); }
}

class C extends P {
    foo() {
        super();
    }
}

var c1 = new C();
c1.foo(); // "P.foo"

var D = {
    foo: function() { console.log( "D.foo" ); }
};

var E = {
    foo: C.prototype.foo
};

// 把 E 委托到 D
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"
```

如果你认为 super 会动态绑定（非常合理！），那你可能期望 super() 会自动识别出 E 委托了 D，所以 E.foo() 中的 super() 应该调用 D.foo()。

但事实并不是这样。出于性能考虑，super 并不像 this 一样是晚绑定（late bound，或者说动态绑定）的，它在 [[HomeObject]].[[Prototype]] 上，[[HomeObject]] 会在创建时静态绑定。

在本例中，super() 会调用 P.foo()，因为方法的 [[HomeObject]] 仍然是 C，C.[[Prototype]] 是 P。

确实可以手动修改 super 绑定，使用 toMethod(..) 绑定或重新绑定方法的 [[HomeObject]]（就像设置对象的 [[Prototype]] 一样！）就可以解决本例的问题：

```
var D = {
    foo: function() { console.log( "D.foo" ); }
```

```
};

// 把 E 委托到 D
var E = Object.create( D );

// 手动把 foo 的 [[HomeObject]] 绑定到 E，E.[[Prototype]] 是 D，所以 super() 是 D.foo()
E.foo = C.prototype.foo.toMethod( E, "foo" );

E.foo(); // "D.foo"
```

> toMethod(..) 会复制方法并把 homeObject 当作第一个参数（也就是我们传入的 E），第二个参数（可选）是新方法的名称（默认是原方法名）。

除此之外，开发者还有可能会遇到其他问题，这有待观察。无论如何，对于引擎自动绑定的 super 来说，你必须时刻警惕是否需要进行手动绑定。唉！

## A.3　静态大于动态吗

通过上面的这些特性可以看出，ES6 的 class 最大的问题在于，（像传统的类一样）它的语法有时会让你认为，定义了一个 class 后，它就变成了一个（未来会被实例化的）东西的静态定义。你会彻底忽略 C 是一个对象，是一个具体的可以直接交互的东西。

在传统面向类的语言中，类定义之后就不会进行修改，所以类的设计模式就不支持修改。但是 JavaScript 最强大的特性之一就是它的动态性，任何对象的定义都可以修改（除非你把它设置成不可变）。

class 似乎不赞成这样做，所以强制让你使用丑陋的 .prototype 语法以及 super 问题，等等。而且对于这种动态产生的问题，class 基本上都没有提供解决方案。

换句话说，class 似乎想告诉你：“动态太难实现了，所以这可能不是个好主意。这里有一种看起来像静态的语法，所以编写静态代码吧。”

对于 JavaScript 来说这是多么悲伤的评论啊：*动态太难实现了，我们假装成静态吧。*（*但是实际上并不是！*）

总地来说，ES6 的 class 想伪装成一种很好的语法问题的解决方案，但是实际上却让问题更难解决而且让 JavaScript 更加难以理解。

如果你使用 `.bind(..)` 函数来硬绑定函数（参见第 2 章），那么这个函数不会像普通函数那样被 ES6 的 `extend` 扩展到子类中。

## A.4　小结

`class` 很好地伪装成 JavaScript 中类和继承设计模式的解决方案，但是它实际上起到了反作用：它隐藏了许多问题并且带来了更多更细小但是危险的问题。

`class` 加深了过去 20 年中对于 JavaScript 中“类”的误解，在某些方面，它产生的问题比解决的多，而且让本来优雅简洁的 `[[Prototype]]` 机制变得非常别扭。

结论：如果 ES6 的 `class` 让 `[[Prototype]]` 变得更加难用而且隐藏了 JavaScript 对象最重要的机制——对象之间的实时委托关联，我们难道不应该认为 `class` 产生的问题比解决的多吗？难道不应该抵制这种设计模式吗？

我无法替你回答这些问题，但是我希望本书能从前所未有的深度分析这些问题，并且能够为你提供回答问题所需的所有信息。

# 版权声明

“Kyle对JavaScript语言每一个细节的缜密思考方式，会潜移默化地移植到我们的头脑和日常工作流程当中。”
——**Shane Hudson**，自由前端开发者

“原型使JavaScript语言功能强大，但也令开发人员备感困惑。本书第二部分‘this和对象原型’精妙地解释了原型、继承和JavaScript中‘类’的概念。”
——**David Walsh**，Mozilla前端工程师

# 你不知道的JavaScript 上卷

JavaScript语言有很多复杂的概念，但却用简单的方式体现出来（比如回调函数），因此，开发者无需理解语言内部的原理，就能编写出功能全面的程序；就像收音机一样，你无需理解里面的管子和线圈都是做什么用的，只要会操作收音机上的按键，就可以收听你喜欢的节目。然而，JavaScript的这些复杂精妙的概念才是语言的精髓，即使是经验丰富的JavaScript开发者，如果没有认真学习也无法真正理解语言本身的特性。正是因为绝大多数人不求甚解，一遇到出乎意料的行为就认为是语言本身有缺陷，进而把相关的特性加入黑名单，久而久之就排除了这门语言的多样性，人为地使它变得不完整、不安全。

“你不知道的JavaScript”系列就是要让不求甚解的JavaScript开发者迎难而上，深入语言内部，弄清楚JavaScript每一个零部件的用途。本书介绍了该系列的两个主题：“作用域和闭包”以及“this和对象原型”。掌握了这些知识之后，无论什么技术、框架和流行词语，你都能轻松理解。

## 作者介绍

**Kyle Simpson** 推崇开放的互联网，对JavaScript、HTML5、实时/端对端通信和Web性能有深入研究。他是技术书作家、技术培训师、讲师和开源社区的活跃成员。

封面设计：Linley Dolby Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机//程序设计/JavaScript

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

oreilly.com
YouDontKnowJS.com

ISBN 978-7-115-38573-4

ISBN 978-7-115-38573-4
定价：49.00元